University of Zurich[UZH]

SOFAS, Software Analysis as a Service

Improving and Rethinking Software Evolution Analysis

A dissertation submitted to the Faculty of Economics,
Business Administration and Information Technology
of the University of Zurich

for the degree of
Doctor of Science

by
Giacomo Ghezzi
from Milano, Italy

Advisors
Prof. Dr. Harald C. Gall, University of Zurich, Switzerland
Prof. Dr. Schahram Dustdar, Vienna University of Technology, Austria

2012

Abstract

Software analysis is one of the key activities in software engineering as it allows to extract the most diverse and extensive information regarding a software system. The classic analyses have been for years the ones targeting models and source code [JR00]. In the last years, many research groups have shifted their attention to software evolution and the community of reverse engineering, reengineering, and program understanding has actually acknowledged that evolution is indeed the umbrella of their research activities. Thus, software evolution analysis is a rather young branch of software engineering. Until recently, historical data was used primarily for historical record supporting activities such as retrieving previous versions of the source code or examining the status of a defect. Studies using this data to analyze various aspects of software development (*e.g.*, software design/architecture, development process and dynamics, etc.), have emerged and flourished only in the last decade. These studies have highlighted the value of collecting and analyzing this data, *e.g.*, to support the maintenance of software systems, improve software design/reuse, and empirically validate novel ideas and techniques. Yet, each of these studies has built its own methodologies and tools to extract, organize and utilize such data to perform their research. Because of this, easy and straight forward synergies between these analyses/tools rarely exist due of their stand-alone nature, their platform dependence, their different input and output formats and the variety of systems to analyze. Therefore, despite this richness, the field still lacks ways to effectively and systematically share, integrate and study data coming from different analyses and providers. We claim that this is vital for the maturing of the field and to expand and boost its role in supporting software development practices.

The key contribution of this thesis is a *distributed and collaborative software analysis platform to enable seamless interoperability of software analysis tools across platform, geographical and organizational boundaries*. In particular, we devise software analysis

tools as services that can be accessed and composed over the Internet. These distributed services are widely accessible through a software analysis broker where organizations, research groups and analysis service providers can register and share their tools. To enable (semi)-automatic use and composition of these tools, they will be classified and mapped into a software analysis taxonomy and adhere to specific meta-models and ontologies for their category of analysis.

First, we introduce the concept of *Software Analysis as a Service* as a way of facilitating access to different analyses from various tools and providers using web services. Along with that, we present two different implementations of *SOFAS* (SOFtware Analysis Services), the architecture implementing this concept. We explain in detail the one we deemed more feasible and appropriate. Such solution follows the principles of a RESTful architecture [Fie00] and allows for a simple yet effective provisioning and use of software analyses based upon the principles of Representational State Transfer around resources on the web. *SOFAS* consists of three main constituents: Software Analysis Web Services (*SA-WS*), Software Analysis Ontologies (*SA-Ontos*) and a Software Analysis Broker (*SA-B*). SA-WS offer different software evolution analyses as standard RESTful web service interfaces. They adhere to specific meta-models and *SA-Ontos* that define and represent the data consumed and produced by them. The *SA-B* acts as the services manager and the interface between the services and the users. It contains a Services Catalog of all the registered analysis services with respect to a specific software analysis taxonomy.

Second, we show in detail how we can adequately and meaningfully describe the software evolution knowledge produced by these services by means of our family of software evolution ontologies called *SEON*. In fact, data itself is not necessarily information, and information is not necessarily knowledge. Successful differentiation requires understanding of data semantics and interpretation. The Semantic Web provides the instruments to solve such dichotomy; ontologies created by human beings represent knowledge and give semantic meaning to raw data so that machines can automatically handle it. Moreover, reasoners make implicit knowledge explicit by inferring relations that were previously missing. In addition to *SOFAS*, we showcase two additional semantics-aware approaches powered by *SEON* that help stakeholders in dealing with large amounts of software evolution data: a natural language query interface for developers and large-scale software visualization.

Third, we present a framework integrated into *SOFAS* for the semi-automated composition of the *SA-WS*. We explain how this composition works and describe SCoLa, a new language we devised to define such composition. We then show two concrete applications of workflows built on top of this framework, used to investigate different aspects of the

evolution of a software system.

We show and validate the effectiveness of our approach using different concrete use cases. *SOFAS* has been used to analyze a family of open source projects and a proprietary system for a software quality audit commissioned by an industrial partner. Its use in this instance has been bifold: answer a specific evolution analysis question and gather a wide range of varied yet interlinked information about the system being studied. In the first case, we answer the question: *"Which are the hotspots and evolution anomalies for a project?"*. In the second, we exploit the data collected to provide several intuitive and interactive visualizations giving different perspectives over the history and quality of the analyzed software. With this use case, we show the effectiveness of *SOFAS* in addressing concrete software quality and evolution analysis industrial needs. In another scenario, we prove how *SOFAS* can be used as a platform to support the replication of empirical studies on software evolution (a long standing issue in the field) and, to a certain extent, more generic software analyses. In particular, we show that we can replicate, to different degrees of completeness, up to 60% of all the empirical studies published at the Working Conference on Mining Software Repositories from 2004 to 2011. At last, the number of existing tools that are already using *SOFAS* is a testimony its usefulness and versatility: SMELL TAGGER [MFGW12], COCOVIZ [BG07] and FACETS OF SOFTWARE EVOLUTION[1] are different software analysis and visualization tools that use *SOFAS* to combine different analyses and extract all the information needed. Without a shared, easily accessible framework such as *SOFAS*, they would have all had to implement the analyses needed from scratch.

In conclusion, drawing from the experience and knowledge gained during this research, we present the blueprint for the version control system (*VCSes*) of the future. Current *VCSes* are, in fact, not built to be systematically analyzed. The only way to extract data from them has so far been the parsing of the bare history logs, which is, in our opinion, far from being the optimal approach. This makes *VCSes* the main bottleneck and source of discrepancies in *SOFAS* and, in general, in software evolution analysis. The system we propose is an analysis friendly plug-in-based *VCS*. Different analyses can be plugged into the system and register for specific repository events. In this way, they can automatically and proactively run and update their data every time it is needed in an incremental fashion.

[1] http://habanero.ifi.uzh.ch:8090/

Zusammenfassung

Der Software Analyse kommt besondere Bedeutung im Software Engineering zu, da mit ihrer Hilfe ein umfassendes Verständnis verschiedenster Aspekte eines Software Systems erlangt werden kann. Traditionell umfasst die Software Analyse Modelle und Programmcode. Je länger, je mehr setzt sich jedoch die Meinung in der Forschergemeinschaft durch, dass neben Modellen und Code auch evolutionäre Aspekte eines Software Systems von Relevanz sind. In Konsequenz hat sich in den letzten Jahren die Software Evolutions Analyse als Überbegriff für Reverse Engineering, Reengineering und Program Understanding etabliert. Software Evolution ist somit ein eher jüngerer Forschungszweig im Software Engineering—Erst im letzten Jahrzehnt wurden historische Daten im Rahmen von Studien verwendet, um verschiedene Aspekte des Software Entwicklungsprozesses (z.B. Software Entwurf und Architektur, die Dynamik des Entwicklungsprozesses, etc.) zu beleuchten. Diese Studien haben gezeigt, dass das Sammeln und Analysieren jener Daten die Grundlage für Fortschritte im Entwurf von Software, in Hinsicht der Wiederverwendbarkeit, Wartung, sowie zur empirischen Validation neuer Ideen und Techniken darstellt.

Im Kontext der besagten Studien wurden stets aufs neue ähnliche Methodiken und Werkzeuge entwickelt, um die benötigten Daten zu extrahieren und organisieren, und sie dann ihrem Forschungszweck zuzuführen. Diese Doppelspurigkeiten erschweren potentielle Synergien zwischen Werkzeugen und Analysen erheblich, da erstere oft plattformabhängig sind und verschiedenste, inkompatible Ein- und Ausgabeformate verwenden. Den Forschern, die sich mit der der Software Evolutions Analyse beschäftigen, mangelt es somit an Mitteln um diese Daten effizient und systematisch auszutauschen, deren Integration voranzutreiben und um sie zu studieren. Wir stellen die These auf, dass dies jedoch eine wichtige Rolle im Reifungsprozess des Forschungsgebietes darstellt und dass eine Vereinheitlichung ein Katalysator für die Akzeptanz in der Software Entwicklungspraxis ist.

Ein wesentlicher Beitrag dieser Doktorarbeit ist die Beschreibung einer verteilten Software Analyse Plattform, welche die nahtlose Interoperabilität mannigfaltiger Software Analyse Werkzeuge über verschiedenste Betriebssysteme, sowie über geographische und institutionelle Grenzen hinweg ermöglicht. Ein besonderer Schwerpunkt wird zudem auf Kollaboration gelegt. Im Speziellen entwickeln wir Software Analyse Werkzeuge nach den Grundsätzen des Service-orientierten Paradigmas, sodass diese im Internet verfügbar und mit einander kombinierbar sind. Diese dezentralisierten Services werden von einem Software Analyse Brokers verwaltet, welcher es Forschungsgruppen und anderen Anbietern von Analyse-Services ermöglicht ihre Werkzeuge zu registrieren und somit öffentlich zugänglich zu machen. Die Werkzeuge werden anhand einer Software Analyse Taxonomie beschrieben. Sie folgen spezifischen Metamodellen und Ontologien, je nach Art ihrer Analyse. Dies erlaubt eine (halb-)automatische Nutzung und Komposition verschiedenster Services.

Gleich zu Beginn dieser Arbeit stellen wir das Konzept der *Software Analyse als Service* als eine Art der Vereinfachung des Zugriffs auf verschiedenste Analysemöglichkeiten durch diverse Werkzeuge und Anbieter mittels Web Services vor. Im Zuge dessen präsentieren wir die Grundzüge zweier verschiedene Implementierungen der *SOFAS* (SOFtware Analyse Services) Architektur. Die vielversprechendere der beiden Implementierungen wird in Folge im Detail beschrieben. Die Lösung folgt dem Paradigma des Representational State Transfer (REST) und gewährleistet so deren einfache, jedoch effektive Verwendung. *SOFAS* besteht aus drei wesentlichen Bestandteilen: Software Analyse Web Services (*SA-WS*), Software Analyse Ontologien (*SA-Ontos*) und dem Software Analyse Brokers (*SA-B*). *SA-WS* repräsentieren verschiedene Software Evolutions Analysen und sind zugänglich über RESTful Web Service Schnittstellen. Die Web Services verwenden spezifische Metamodelle und *SA-Ontos* zur Beschreibung ihrer Ein- und Ausgabeparameter. Der *SA-B* agiert als Service Manager und als Schnittstelle zwischen den Services und dem Benutzer. Er enthält einen Service Katalog mit Beschreibungen aller registrierten Analyse-Services. Diese Beschreibung basiert auf der eingangs erwähnten Software Analyse Taxonomie.

Im weiteren Verlauf der Arbeit erklären wir, wie man jenes Wissen über verschiedene Aspekte der Evolution von Software, welches durch unsere Services generiert wird, adäquat anhand einer Familie von Software Evolutions Ontologien namens *SEON* beschreiben kann. Tatsächlich sind Daten nicht notwendigerweise mit Information gleichzusetzen und Information nicht zwingend mit Wissen. Eine sinnvolle Unterscheidung bedingt jedoch tiefergehendes Verständnis und Interpretation der Daten. Das Semantische Netz stellt

geeignete Werkzeuge bereit, um solch einer Dichotomie zu begegnen; Ontologien, durch Menschenhand erschaffen, repräsentieren Wissen und versehen Rohdaten mit Bedeutung, sodass Maschinen das Wissen automatisiert verarbeiten können. Reasoner machen implizites Wissen explizit, indem sie fehlende Beziehungen automatisch ableiten. Neben *SOFAS* stellen wir zwei weitere auf *SEON* basierende Ansätze vor, um verschiedene Akteure im Umgang mit grossen Mengen an Software Evolutions Daten zu unterstützen. Die beiden Ansätze umfassen eine natürlich sprachige Benutzerschnittstelle für Software Entwickler, sowie eine Visualisierung von Software.

Schliesslich präsentieren wir ein Rahmenwerk zur halb-automatischen Komposition der *SA-WS*. Sowohl die Funktionsweise der Komposition, wie auch *SCoLa*– eine Sprache zur Beschreibung der Komposition – werden im Detail beschrieben. Als nächstes zeigen wir zwei konkrete Anwendung von Workflows, die mittels des Rahmenwerks realisiert werden um verschiedene Aspekte der Evolution eines Software Systems zu untersuchen.

Wir illustrieren und validieren die Effektivität unseres Ansatzes mittels mehrerer konkreter Anwendungsfälle. Wir nehmen *SOFAS* her und analysieren eine Familie von Open Source Projekten, sowie ein proprietäres System. Letztere Analyse wurde im Rahmen eines Software Qualitäts Audits im Auftrag eines Industriepartners durchgeführt. Die Ziele der Analyse umfassten das Auffinden von Hotspots im Code und Anomalien in der Evolution des Systems, sowie die Bereitstellung von intuitiven und interaktiven Visualisierungen, welche verschiedene Sichten auf die Entwicklungsgeschichte und Qualität der analysierten Software bieten. Während dieser Anwendungsfall die Effektivität von *SOFAS* illustriert, demonstrieren wir anhand eines weiteren Szenarios, wie *SOFAS* als Replikations-Plattform für empirische Studien über Software Evolution, respektive Software Analyse, dienen kann. *SOFAS* erlaubt es uns bis zu 60% aller empirischer Studien zu replizieren, welche zwischen 2004 und 2011 im Rahmen der Working Conference on Mining Software Repositories vorgestellt wurden. Darüber hinaus bestätigt die wachsende Zahl bereits vorhandener Werkzeuge, die auf *SOFAS* aufbauen, dessen Verwendbarkeit und Vielseitigkeit. Beispiele für solche Werkzeuge zur Analyse und Visualisierung von Software sind SMELL TAGGER [MFGW12], COCOVIZ [BG07] und FACETS OF SOFTWARE EVOLUTION[2]. Die Entwicklung dieser Werkzeuge hätte sich ohne das Rahmenwerk, das *SOFAS* bereit stellt, deutlich aufwendiger gestaltet.

Basierend auf der Erfahrung und dem Wissen, welches im Rahmen dieser Doktorarbeit gewonnen wurde, präsentieren wir abschliessend einen Entwurf für ein Versionierungssystem (*VCS*) der nächsten Generation. Bestehende *VCS* weisen Defizite hinsichtlich der

[2]http://habanero.ifi.uzh.ch:8090/

systematischen Analysierbarkeit der gespeicherten Informationen auf. Die Extraktion der benötigten Daten gestaltet sich üblicherweise schwierig und läuft oft auf das Parsen von Log-Dateien heraus. *VCS* stellen daher einen Flaschenhals und eine Quelle der Diskrepanz im Kontext von *SOFAS* im Speziellen und in der Software Evolutions Analyse im Allgemeinen dar. Das von uns vorgeschlagene System ist Analyse-freundlich und basiert auf einer Plug-in Architektur. Verschiedene Analysewerkzeuge können einfach hinzugefügt und für bestimmte Ereignisse registriert werden. Die Werkzeuge können so automatisch und pro-aktiv ausgeführt werden, sodass sie auf inkrementelle Art und Weise die von ihnen generierten Daten aktuell halten können.

Contents

List of Figures

List of Tables

List of Listings

1

Synopsis

1.1 Introduction

Successful software systems must be continuously maintained and adapted or they become progressively less useful. However, this constant evolution increases the complexity and the amount of disorder in the system, leading to a slow but continuous deterioration in quality and usability. Such process is also known as *software entropy* [LB85], *code decay* [EGK+11] or *software aging* [Par94]. The constant changes, essential for the success of software, are at the same time one of the main causes of its dismissal. As a consequence, more resources are needed to preserve and simplify its structure. Software maintenance and evolution thus account for a large portion of the development effort and cost. According to McKee [McK84], it makes up for two thirds of the total software development effort. De Roze *et al.* [RN78] quantified maintenance costs as being one third of the total development costs, while other studies estimated that they exceed 50% of the total costs, e.g., [Boe76, PA06], and are continuously increasing. More recent studies even claim that maintenance represents up to 90% of the entire software life-cycle costs [Erl00].

Having an always up to date and thorough view of a software system, its health and history greatly helps in mitigating this problem and reduce the costs. Historical data stored into repositories such as version control, bug and issue tracking, or mailing lists is essential for that purpose. Until recently, this data had been mostly neglected or considered just

a necessary byproduct. However, studies have highlighted the value of collecting and analyzing this diverse source of data. This sparked what can be considered a "gold rush" to mine all sorts of useful information. A growing number of analysis techniques, such as static and dynamic code analyses, code clone detection, co-change analysis, bug prediction or hot spots detection, have been devised. Yet, despite this richness, the issue of easy and straight forward integration and sharing of data produced by different analyses has been left almost entirely unaddressed. Each of these techniques relies on its own methodologies and tools to extract, organize and utilize such data to produce the results needed by the addressed stakeholder. For every analysis, a specialized tool, with its own explicit or implicit meta-model dictating how to represent the input and the output, has to be installed, configured and executed. As a result, the sharing of information between tools is only possible by means of a cumbersome export towards files complying to a specified exchange format. Even if different analyses of the same kind exist (*e.g.*, code duplication analysis), there is no way to compare their results or integrate them other than manual investigation. Interoperability is hampered even more by the stand-alone nature of such analyses, as well as their platform and language dependence.

The use and combination of different software analysis tools is still a challenging problem when trying to gain a deeper insight into the history of a software system. A critical assessment of the research fields uncovers the fact that people, instead of reusing and taking advantage of each other's work, keep re-inventing the wheel with little advancement of the field as a whole. Moreover, the replication of empirical studies on software evolution is negatively affected. In fact, as shown by Robles [Rob10], both the analyses and the results, even when available, are rarely usable for replication in an effective way. Because of this, even though software evolution research has a strong foundation on empirical studies, a systematic framework enabling replicability is still missing. We claim that this status quo severely hampers the progress of software evolution research and its soundness.

The goal of this thesis is to solve this problem by devising a *distributed and collaborative software analysis platform to allow for interoperability of software analysis tools across platform, geographical and organizational boundaries*. In order to do so, we investigate the problem in detail, propose and implement a feasible and comprehensive solution and demonstrate its potential and usefulness. This involves answering the following four main research questions:

Research Question 1 (RQ1) How can we offer the functionalities of this plethora of evolution analyses in a consistent and efficient way?

Research Question 2 (RQ2) How can we effectively describe the heterogeneous and wide-ranging data produced by such analyses?

Research Question 3 (RQ3) How can we (semi)-automatically compose these analyses to provide more complex higher-level analyses?

Research Question 4 (RQ4) What is the impact of our proposed platform on the field of software evolution analysis?

We claim that our solution can enhance and speed up the work of a software engineer by giving her access to a wide amount of information without the need to install several tools and to cope with many output formats. It will also promote the uncovering of new, meaningful and interesting metrics deriving from the most diverse types of analysis that can finally "talk" to each other or can be combined. At last, such a platform could also play a role in facilitating the replication of empirical studies by making analyses and their results easily available.

Before proceeding, we briefly illustrate the challenges we want to address and the potential impact of our work with three scenarios on: (1) software analysis collaboration, (2) software analysis data usage, and (3) software evolution study replication. Each scenario description is followed by an explanation about how our proposed approach should help in its resolution. For these scenarios we consider three software evolution analysts: Alice, Charlie and Bob; a physicist, Jack, interested in the open source projects as complex systems; and an empirical software engineer, Jill.

Software Analysis Collaboration

The Scenario Alice has developed a tool extracting the detailed CVS history of software projects to gain better insights on the development process. Bob has a tool doing the same but with Bugzilla data, and Charlie's tool extracts the FAMIX model of a given object-oriented software project by parsing its entire source code to obtain an unambiguous and precise language independent representation of it. Each of the tools works on a specific platform and requires its own settings and parameters. Alice, Bob, and Charlie do not work for the same institution. They decided to unify their efforts to throughly analyze the history of Foozilla, a multi-million lines of code system, but the communication overhead due to different data-models, different result data formats, and storage media are too cumbersome to follow-up on their exciting plan. So they start thinking of a unified software analysis platform that

would allow them to easily get a detailed holistic view of the history of Foozilla: it would tell them for example for each release which bugs are related to specific files revisions, thus providing a clear link between a bug and some specific source code files. Bug prone parts, bug fixing and other source code change patterns would then be easy to spot. Based on this, new and additional analyses could be produced and offered on the same platform. For example, Jane could then develop the analysis she always wanted to implement but lacked the right expertise and tool support. That analysis calculates source code metrics (through its FAMIX model, without thus having to deal with the actual source code) for each CVS release to spot code smells to both show their trend over time and their relation to reported bugs and eventually show that into some nice navigable graphical interface.

Our Solution Our approach should offer Alice, Bob and Charlie the platform they need to share and combine their tools. Such a platform would then act as an "analysis portal" for software evolution researchers (not just for Alice, Bob and Charlie) through which they can share their analyses and use each other's analyses with only little overhead. These analyses would all offer a uniform interface that facilitates their use and composition. Moreover, such a composition would be supported by appropriate interfaces for human users and applications. These interfaces would support the (semi)-automated combination of analyses into complex workflows.

Software Analysis Data Usage

The Scenario Jack, a physics researcher studying complex systems, is interested in studying open source projects. These systems are a perfect target for many complex systems/networks empirical studies as they themselves are complex self-organizing adaptive systems and all their related data (from the code release history to the mailing list discussions) is stored and readily available to everyone. In particular he wants to rigorously assess, using a broad and varied sample of big open source projects, whether laws that in the years have been found to be common in complex systems, such as Preferential Attachment, n-degrees of separation, Zipf's Law, etc., apply also to object oriented software systems. To start off, he would like to quickly check whether proportional growth, and Zipf's power law as well, apply to the growth of inter-class dependencies. For this, he would need to calculate, for every release of all the systems being studied, the in and out degrees of every class. Given his background and time constraints, he cannot do all those calculations by hand nor

he has the knowledge to write an application that does that automatically.

Our Solution Our approach should offer an online accessible platform, were outsiders can be guided in the use and combination of a wide spectrum of analyses to exploit software evolution data for the most disparate and unforeseen purposes. In this specific case, Jack would need to combine an analysis that extracts the version control history of a software project with one extracting a detailed source code model for every release detected in the previous step. Given a sufficiently detailed source code model (e.g. FAMIX [TDD00]), counting the in and out degrees for each class is then trivial.

Software Evolution Study Replication

The Scenario Jill wants to replicate an empirical study on bug prediction recently published at MSR (Working Conference on Mining Software Repositories). In particular, she would like to check whether the main findings (the time of the day and the day of the week influence the introduction of bugs and regularly committing developers introduce less bugs) hold for a wider spectrum of projects that did not feature in the original study. In order to do that, she would either need to develop her own analysis, based on the algorithm reported on the paper, or re-use the tool developed by the authors of the original study. As we already said, even when the original tools are available, they are seldom re-useable in an effective way.

Our Solution Our approach should help solve Jill's issues by offering the analyses needed for the replication (or conceptually similar ones) through an online platform. Moreover, the results should also be available online directly from the analysis itself, following explicit, well-defined models. In this way, the results and claims of the original study could be easily verified and compared with the results of any replication.

1.2 Software Evolution Analysis

Software analysis is one of the key activities in software engineering, as it allows to extract the most diverse and extensive information regarding a software system, *e.g.*, for the purpose of evolution analysis, reengineering, etc. The classic analyses targeting models and source code have been around for quite some time [JR00]. In the last years many research

groups have shifted their attention to software evolution and the community of reverse engineering, reengineering, and program comprehension has actually acknowledged that evolution is indeed the umbrella of their research activities. A wide variety of analyses have thus been proposed and implemented throughout the years. More recently, faced with such a diverse assortment of useful, yet independent, analyses researchers have come up with several software data exchange languages. To further facilitate data exchange and analysis combination, varied platforms and data repositories have been devised.

In the following, we give an overview on: (1) the most notable work on software (evolution) analysis; (2) the existing software data exchange languages; and (3) the platforms and data repositories proposed.

1.2.1 Software (Evolution) Analysis

Software consists of two main cornerstones: process and product. The process is the structure imposed on the development of a software product. It comprises the policies, organizational structures, technologies, procedures, and artifacts that are needed to design, develop and maintain such a product. Thereby, the software product is the final, tangible outcome of that process. Software analyses always target one of these two aspects. The ones tackling process, often also called *software development analyses*, focus on three main dimensions, i.e. the development history (extraction, prediction and analysis of source code changes and bugs), its underlying process and the people involved in it. Analyses studying the product focus on its underlying models or the actual source code. Model analysis targets the extraction of specific behavioral and structural model representations, the differencing between two models, usually of two versions of the same system. Code analysis, being the oldest and thus most studied topic of this taxonomy, targets a wide range of categories. For example, code well-formedness, correctness and quality. The last category includes, among others, code security, conciseness, performance and design. Due to space limitations and to the impressive body of research done, we will mainly focus on software process analysis, as it is the one targeting software evolution, this thesis's field of interest.

Software Development Analysis

There is an abundance of research works and tools exploiting software project data for historical software analysis. This data is usually extracted from software repositories

such as version archives, issue tracking systems or archived communication (newsgroups, mailing lists, chat logs, etc.).

Different works address the extraction and combination of data coming from different repositories to facilitate its subsequent analysis. Fischer *et al.* combine version control and bug tracking information (*i.e.*, CVS and Bugzilla) into a release history database [FPG03b], to support co-change [GJK03] and fine-grained source code change [GFP09] analyses. SoftChange [Ger04] integrates also information extracted from mailing lists to calculate evolution statistics and to create several visualizations to aid the exploration of a software history and the discovery useful, unknown facts (e.g. the social network of the developers). Draheim *et al.* [DP03] and Bevan *et al.* [BWKG05] propose similar approaches, but only focus on version control data. Hipikat [uM03] combines CVS source repository data, Bugzilla data, messages posted on developer forums and other project documents to support the recommendation of artifacts that are relevant to a particular development task.

Change analysis

The information about the changes performed on artifacts during a software project's lifetime has been exploited to uncover a wide range of additional knowledge on such changes. For example, co-change analysis (the detection of clusters of artifacts that frequently changed together) has been used to uncover hidden dependencies and relationships between artifacts [GHJ98], identify problematic code entities [BW03] or predict further code changes and recommend potentially relevant source code for a particular modification task [ZWDZ04, YMNCC04]. However, the information about code changes that can be directly fetched from version control systems is extremely coarse grained. As a matter of fact, it only consists of the names of the affected files and the lines that were changed. Several works thus focus on extracting and inferring more detailed information from this crude data. For example, Zou *et al.* [ZG03] use origin analysis to detect the merging and splitting of code entities, while S. Kim *et al.* [KPW05] use it to track function name changes. M. Kim *et al.* [KSNM05] used such data to track the evolution of code clones and build a clone genealogy. On the other hand, Fluri *et al.* [FWPG07] use tree differencing to extract several source code change types and classify them based on their significance. Such significance is assessed in terms of the impact of the change on other code entities and on functionality (functionality-modifying vs. functionality-preserving). M. Kim *et al.* [KN09] follow a logic rules-based approach to detect such changes. Other studies aim at extracting much more high-level information, such as refactorings [APM04, DDN00, WD06, PRSK10] or patterns and their violations [LZ05]. At last, change information is also used to predict

future faults. For example, Nagappan *et al.* [NB05], extract a set of code churn metrics to predict the defect density of a system and discriminate between fault and not fault-prone entities. Giger *et al.* extract and test different types of information to predict such faults. In [GPG11a] they use fine-grained source code changes and show that they outperform code churn in fault prediction. In [GPG11b] they measure code ownership using the Gini coefficient and show that fewer bugs can be expected if a large share of all changes carried out by relatively few developers.

Bug analysis

Similarly to the works on source code change, bug analysis addresses the extraction of data from a bug repository (as we already saw in [FPG03b]), its analysis and the prediction of future bugs. Based on the experience gained from the analysis of this data, Bettenburg *et al.* [BJS+08] propose CUEZILLA, a tool to measure the quality of new bug reports and to recommend quality improvements. Anvik *et al.* [AHM06] apply a machine learning algorithm to the same data to learn the kinds of reports each developer resolves and use such information to improve the bug triaging process. To help this process, Jeong *et al.* [JKZ09] reconstruct the bug tossing (a.k.a. reassignment) history to discover team structures, find suitable experts for a new task and thus better assign new bugs to the most appropriate developers. On the other hand, Weiss *et al.* [WPZZ07] and Giger *et al.* [GPG10] use attributes of past bug reports to predict the probable time effort needed for the resolution of new submitted bugs. The former only use the resolution time for such prediction, while the latter use a combination of bug report attributes. Bug data has also been combined with version control history to detect fault prone locations. For example, Hassan *et al.* [HH05] use a dynamic cache of the ten most error prone directories (the ones that were changed during bug fixes). Similarly, Kim *et al.* [KZJZ07] and Sliwerski *et al.* [SZZ05a] propose a dynamic cache of the most error prone source code locations. They cache the location of the fault itself and any other locations in which code was changed during the fix. By consulting this cache when a fault is fixed, a developer can then detect additional, possibly unknown fault-prone locations. Sliwerski *et al.* also use this information to annotate these fault prone locations with color bars in Eclipse. The use of such caches is useful to better allocate resources and prioritize testing efforts on the most problematic parts of a software.

Development dynamics analysis

Not only the history of a software development process has been addressed, but also its

underlying dynamics. In particular, the role of the developers in evolutionary processes. Mockus *et al.* [MH02], Girba *et al.* [GKSD05] and Minto *et al.* [MM07] locate people with desired expertise/ownership by analyzing version control history data. In addition to that, Girba *et al.* extract different behavioral patterns, *i.e.*, when and how different developers interact, in which way and in which part of the system. Code ownership information is then used to better assign tasks, bugs or find the developers who know a piece of code the better. Rahman *et al.* [RD11] and Bird *et al.* [BNM+11] further exploit this information by combining it with bug history data to better understand their relationship and influence on software development (e.g. do the number of contributors involved, the developer's experience or the ownership distribution matter?). Another way to study the dynamics of software development is to analyze the social network of the involved developers. Such information is mostly extracted from mailing lists and message boards. Bird *et al.* [BPD+08] use it to detect the structure of the social network, the different subcommunities that spontaneously arise within software projects and their different structures. In another study, Bird *et al.* [BGD+07] investigate the evolution of the social network to analyze how users join different open source projects. Wagstron *et al.* [WHC05] combine it with information extracted from blogs and networking websites to build models of the social behavior of developers. Begel *et al.* [BKZ10] capture the relationships between people, code, bugs, specifications, and several other work artifacts by mining different Microsoft's internal repositories (version control, mailing list, issue tracking, web sites, etc.). With this information, they are able to discover, track and maintain connections between people and their associated work artifacts to better support inter-team coordination and communication.

Design Analysis

The reverse engineering of a wide range of abstractions and forms of representations from software systems is a fecund and vital software analysis field. These analyses can either be static or dynamic. Static analysis does not account for program input; the result is thus applicable to all executions of the program. In contrast, dynamic analysis takes program input into account. A lot of work has been done to extract UML documentation of a system both from its source code and from its runtime behavior, *e.g.*, UML Collaboration Diagrams [KG01,TP03], UML Sequence Diagrams [RC05], UML State Machine Diagrams [LBL06, Sys00] or UML Sequence Diagrams [GZ05, LBL06]. Many UML modeling tools

also allow reverse engineering of Class Diagrams, such as open source ones like Fujaba [1] and ArgoUML [2] or commercial ones like Together [3] and IBM Rational Rose [4]. Apart from UML, many other metamodels and custom representations have been developed during the years and, along with them, tools to infer them from existing software systems. For example, Rigi [MK88] and FAMIX [TDD00] are used to describe the static structure of source code. ACME [GMW00], on the other hand, describes a system's architecture based on its dynamic behavior [SAG+06].

Other works identify design patterns in the source code to promote reuse and assess code quality. Also in this case, both static techniques [ACPF01, GSZ04, KP96, THCS06] and dynamic techniques [HHHL03] have been used. All the techniques are based on cliché matching either on execution traces (for dynamic approachess) or at source code level (for static approaches). At last, a lot of work on extracting call graph has been done, both dynamically [VRHS+99, Mar94, GAM96], and statically [htt]. In particular we want to mention the work by Lhoták *et al.* [Lho07] that proposes a metamodel for unambiguous representation of call graphs and combined dynamic and static analysis. There are also some works that combine model analysis and software evolution, offering design differencing. For example, Xing *et al.* [XS05] present an algorithm that automatically detects structural changes between the design of two versions of a software (fetched from a CVS repository), by analyzing their reverse engineered UML Class Diagrams.

Code Analysis

The analysis of the source code itself is a thriving research field and it is one of the oldest software engineering activities. Started in the compiler community, way before software configuration management systems and any notion of software model existed, it has spread into a variety of software engineering tasks, such as the assessment of code correctness, design quality, well-formedness, security, performance, etc. Like model analysis and extraction, also these analyses can either be static or dynamic.

A lot of successful research has been dedicated, for example, to detect and track clones in software systems. The outcome was the development of different techniques, i.e., token-based [KKI02, Bak95], AST-based [BYM+98], metrics-based [MLM96]. Empirical studies have then combined this information with historical data to analyze the evolution of

[1] http://wwwcs.uni-paderborn.de/cs/fujaba/

[2] http://argouml.tigris.org/

[3] http://www.borland.com/us/products/together/

[4] http://www-01.ibm.com/software/awdtools/swarchitect/

clones in large software systems. The impact of clones on code quality and maintainability is still being discussed as it highly depends on the case studies. Related to code clone analysis is the extraction and calculation of a vast range of metrics from source code. These metrics range from the simple lines of code count (LOC) to more complex ones such as module cohesion and coupling [SMC74], Halstead's metrics [Hal77] or Mc Cabe's Cyclomatic complexity [McC76a]. Their uses span from assessing generic code quality [LM05] (as code clones) to fault prediction [GFS05, ZNG+09] and measuring developers productivity [Jon78]. They have also been combined with evolutionary data [PGFL05] to get a better grasp on how systems evolve and to highlight possible design issues.

1.2.2 Analysis Data Representation

Throughout the years, several generic formats have been proposed to solve the issues of data exchange between tools. One of the earliest format is CDIF (CASE Data Interchange Format) [Imb91], a flat text-based language for exchanging data between CASE tools. Its widely used XML-based successor XMI (XML Metadata Interchange) [Xmi07] is an OMG standard capable of describing different models and even graphics. It is commonly used to pass models created in modeling tools to code generation tools. However, many tool providers extend XMI with proprietary elements, leading to the erosion of the standard. GXL (Graph eXchange Language) [HWS00] is an effort to create a common graph interchange format encompassing TA [Hol98], TGraphs [EF95], RPA [FKO98], RSF [MK88], and PROGRES [SWZ99] to be used by graph based tools.

A number of meta-models describing the structure of software source code of a software at different levels of abstraction have also been devised. The Dagstuhl Middle Metamodel (DMM) [LTP04], captures and describes program level entities and their relationship for both object-oriented and procedural languages. Similar to this, but focused on specific languages, are the C++ Data Model [CGK98] of Chen–describing source code written in C++–and the FAMOOS Information Exchange Model (FAMIX) of Tichelaar et al. [TDD00], describing source code written in any OO language.

Despite the advent of these specialized exchange formats, data is often still serialized into plain XML or a comma separated value (csv) format. These formats are not semantics-preserving and therefore of limited use, as a lot of information is lost in such transformations. To overcome this problem, several researchers have proposed the use of Semantic Web Technologies. These works mainly focus on providing ontologies to represent software analysis data and concepts to foster software reuse and maintenance. Oberle

et al. point out that the domain of software is a primary candidate for being formalized in an ontology [OGS09], by being sufficiently complex and reasonably stable in paradigms and aspects. Consequently, they present a reference ontology for the software engineering domain used to distinguish fundamental concepts such as, data and software. Hyland-Wood *et al.* [HWCK06] propose an OWL ontology of software engineering concepts (SEC), including classes, tests, metrics and requirements. Bertoa *et al.* [BVG06] focus on software measurement. Happel *et al.* [HKST06] in their KOntoR approach store and support the querying of higher level meta-data about software components, such as the programming language, licensing models, ownership and authorship data. Witte *et al.* [WZR07] use text mining and static code analysis to map documentation to source code for software maintenance purposes. These mappings were represented in RDF. Dietrich *et al.* propose a tool that scans the abstract syntax tree of Java programs and detects design patterns for documentation purposes [DE05]. The design patterns are described in terms of OWL ontologies.

Researchers also used OWL ontologies to describe software evolution artifacts found in software repositories. Kiefer *et al.* [KBT07] describe and integrate different artifact sources (source code, issue tracking, version control) into an ontology called EvoOnt, to facilitate common repository mining. Tappolet *et al.* [TKB10] show that EvoOnt could be effectively used to facilitate many of the several software evolution analysis experiments from previous Mining Software Repositories Workshops. At last, Iqbal *et al.* with their *Linked Data Driven Software Development* (LD2SD) methodology, transform software data stored in different repositories (JIRA bug trackers, Subversion, developer blogs, project mailing lists, etc.) into the RDF format to provide a uniform and central RDF-based access to such heterogeneous data.

1.2.3 Analysis Platforms and Data Repositories

There is a wide range of research works exploiting software project data for software evolution analysis and many others proposing different mediums to describe and share such type of data. However, these two research directions rarely intersect. That is, concrete analyses seldom address the sharing and re-use of their results (and of themselves altogether), while approaches aimed at facilitating such reuse are rarely used by concrete analyses. To try to fill this gap, different online analysis platforms and data repositories have been proposed.

Drawing inspiration from the PROMISE repository [GBO07], researchers have es-

tablished several on-line software evolution data repositories in the recent years. Flossmole [HCC06] is probably the biggest and most notable one, containing nearly 1 TB of data extracted from all the major code source forges (sourceforge, github, freshmeat, etc.), covering the 2004-now period. This includes the metadata of more than 500,000 open source projects, automatically scraped from their forge web page. Such metadata includes: the programming language(s) used, the platform(s) supported, the license, the number of developers and brief information about them (name, username, email), developer's role on the project, issue-tracking data (e.g. date opened, status, date closed, and priority), etc. All this data focuses on more high-level development information and dynamics and is offered "as-is". No actual analysis is performed, nor the project source code is investigated. The Ultimate Debian Database [NZ10] follows a similar approach but only for the Debian Linux distribution and all its binary packages. However, it focuses on extracting and presenting more system specific information (package popularity, history of packages upload, etc.) to try to countermeasure the lack of a proper development infrastructure that other Linux distributions (e.g. Red Hat and Ubuntu) have. Mockus [Moc09], on the other hand, collect and index the version control history and the actual source code of a large sample of software projects from the most notable forges. The author discusses the methods developed to build such dataset, but the actual data is not publicly available.

Through these data repositories, varied historical software data can be easily shared and retrieved on-line. However, they do not allow to proactively fetch such data for new projects, as they only handle a fixed, pre-defined set of projects picked by the repository creators. This lack of proactiveness is overcome by tools and platforms combining a wide range of software analyses. Some of these tools, Kenyon [BWKG05], Evolizer [GFP09] and softChange [Ger04], were previously introduced in Section 1.2.1. With them researchers can automatically extract and combine historical data, such as source code change history, bug history and additional analyses (e.g., fine grained source code change extraction, source code meta-model reconstruction, etc.). However, as we argued, none of them make the results, nor the analysis, easily available from outside of the tool. Online analysis platforms, as pointed out by Gasser et al. [GRS04], would exactly fill this gap. Gasser *et al.* also outline a general blueprint for such platforms, which is actually implemented by systems such as FOSSology [Gob08] and Alitheia Core [GS09]. FOSSology is a framework to analyze source code with different, custom analyses (called agents) that can be created by users to fulfill their specific needs. The framework itself is just in charge of extracting the source code from a given repository which will then be analyzed by these analysis agents. However, as of now, the framework only presents an agent that detects

the code license. Alitheia Core is an extensible platform for software evolution analysis, integrating data collection facilities (from software repositories) with a varied assortment of analyses making use of that data. Its main intention is to foster the creation of an extensible ecosystem of shared analyses and results for researchers to enrich and exploit. It is based on a custom OSGI-backed architecture in which the different analyses have to be implemented as custom plugins. The analyses and their results are available through a web interface. While plugins introduce a good amount of flexibility in what the analysis offered are, it is still a highly centralized approach. That is, if another analysis has to be added, it has to be developed as an Alitheia-specific plugin and manually integrated into the system. To put it in a nutshell, it offers functionalities similar to Evolizer or Kenyon, but exposes them also through a web interface for easier access.

So far, Jin and Cordy [JC05], with their Ontological Adaptive Service-Sharing Integration System (OASIS), are the only researchers who proposed a lightweight platform to integrate different software analysis tools offered by different providers. They use a service-sharing methodology that employs a common domain ontology defining the conceptual space shared by the different tools and specially constructed external tool adapters, that wrap the tools into services. A proof of concept with three existing reverse engineering was implemented to explore service-sharing as a viable means for facilitating interoperability among tools.

1.3 The Need for a Different Approach to Software Evolution Analysis: Motivation and Thesis Goal

The approaches presented in Section 1.2.1 allow one to extract a remarkable amount of varied historical software data. The usefulness of such data has also been proved by a number of empirical studies. The success of the Working Conference on Mining Software Repositories (MSR), which mainly deals with the extraction and analysis of that data, is a testament to that. The languages and meta-models we presented in Section 1.2.2 have been devised to make sense of such a wide ranged corpus of information and to consolidate it into known, well-structured, easily shareable representations, languages and meta-models. However, there is still a wide gap between these two research directions (*i.e.*, the actual analyses and the data representation). Languages and meta-models have rarely been used by concrete analyses, outside use cases and proof of concepts. On the other hand, the issue of sharing and integrating data–and the analysis itself–has been ignored by the analyses and approaches proposed throughout the years. The data repositories and platforms we presented in Section 1.2.3 represent some of the main efforts addressing those issues. Data repositories make all sort of software related historical data easily available online. Online platforms expose the analyses and their results online, using different interfaces. However, both approaches do not use any of the existing languages to describe such data, but they all use their own custom, often implicit, meta-models. Many times, these models are a basic transliteration of the data model of the databases in which the data is stored. Furthermore, the existing platforms do not make provision for the programmatical use of the exposed analyses through standard interfaces, e.g. web services, RMI, sockets, etc. In fact, the analyses are often hidden behind custom UIs to facilitate their use by human users.

Therefore, software evolution analyses still suffer from three main problems. They are *rarely easy to re-use*, they exhibit *lack of clear and uniform data representation* and have *insufficient support for straight forward integration.*

Analyses are rarely easy to use. Analyses are more often than not developed as prototypes to be used only within the same research group or even just for a single empirical study. Moreover, as pointed out by Robles [Rob10], such tools are seldom publicly available. Their use by a third party is hampered even more by their stand-alone nature, as well as their platform and language dependence.

Analyses lack a clear and uniform data representation. Since the sharing of results is rarely a concern, analyses have their own explicit or implicit meta-model which dictates how to represent the input and the output data. Thus the sharing of information between tools is only possible by means of a cumbersome export towards files complying to a specified exchange format. Even if analyses of the same kind (e.g., code duplication analysis) exist, there is hardly any way to compare the results or integrate them other than manual investigation.

No straight forward integration between different analyses exists. The two problems we just outlined are the main reasons behind the lack of any systematic way to combine different analyses or their results. This significantly restrains the usage and reduces the acceptance of software evolution analysis tools both by tool builders that would otherwise greatly benefit from that huge amount of diverse information.

The goal of this thesis is to overcome problems of analysis usage, data representation and sharing and analysis integration by devising a distributed and collaborative software analysis platform to allow for interoperability of software evolution analyses across platform, geographical and organizational boundaries. Analyses expose their functionalities and data through a common web service interface and are mapped into a software analysis taxonomy. According to their category, they adhere to specific ontologies describing their input, output and the analysis itself. Their uniform interface enables their use over the Internet and their semi-automatic composition into complex analysis workflows. Such workflows can be used to ask specific questions regarding the evolution and the quality of software or to extract a wider range of data to fulfill broader and more open-ended information needs. Moreover, these workflows and the single analyses can be also used to replicate existing empirical studies on software evolution.

1.4 Foundation and Structure of the Thesis

This thesis has had, since the beginning, a strong engineering and implementation foundation. Its main contribution is the concept of *Software Analysis as a Service* that we devised to solve the research problems we previously outlined, and the RESTful platform implementing it, called *SOFAS* (SOFtware Analysis Services), which at the time of writing is made up of more than 300000 lines of code (mainly Java). This consists of more than 25 different analyses services and their webpages, the core common infrastructure and a web UI offering human users a more intuitive access to *SOFAS*. The core common infrastructure and the web UI is what the users will perceive as *SOFAS*, acting as the interface between them and the services, implementing all the functionalities needed to track, catalog, compose and use them.

The description of *SOFAS* is split between two separate works. In the first one we present its entire core architecture, its considerations and implementation aspects. In the second one we present in detail its software analysis composition component, together with the custom composition language we developed to define such composition. This work also presents a use case validation of *SOFAS* and its composition facilities. The solution we conceived to describe–using Semantic Web technologies–in a uniform and versatile way the data produced and consumed by the different software analyses is presented in a dedicated research work. In such work we also illustrate the usefulness of such a solution in other software engineering fields. At last, the entire approach is validated by proving its effectiveness in replicating existing empirical software evolution studies. Such novel combination of RESTful web services and Semantic Web to solve the issues outlined previously in Section 1.3 and answer the thesis's research questions is depicted in Figure 1.1. As the figure shows, *SOFAS* is the product of the intersection of Semantic Web, REST and software evolution analysis. Moreover, the figure outlines the main concepts of these three constituents playing a part in such intersection.

In the remainder of this section, we briefly summarize the works forming the thesis. For each of them, we (1) describe the specific problem they target, (2) summarize their contribution to the overall thesis statement, (3) outline how they were evaluated (if an evaluation was carried out) and, (4) specify the research question(s) we previously outlined (see Section 1.1) that they answer (if any were answered). The complete studies themselves are presented in Chapters 2-8 (see Section 1.6).

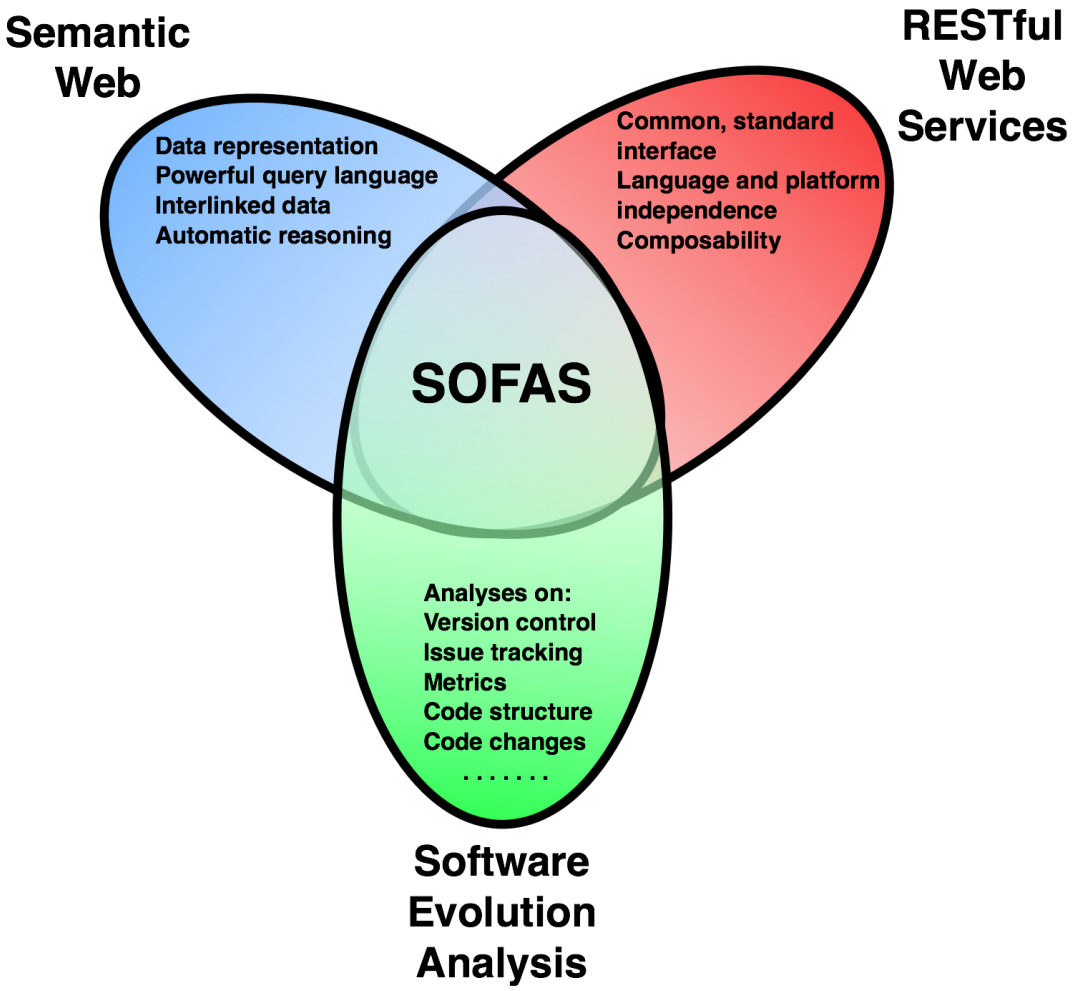

Figure 1.1: The combination of RESTful web services, Semantic Web and software evolution analyses making up *SOFAS*

1. ***Software Analysis as a Service.***

   Problem. Software evolution analysis is still plagued by a plethora of issues. In particular, analyses are rarely easy to re-use, exhibit lack of clear and uniform data representation and have insufficient support for straight forward integration. We need to rethink software evolution analysis.

   Contribution. This study represents the very foundation of this entire thesis, as we frame for the first time the core research issues motivating the thesis. In this

study, we introduce the founding concept of *Software Analysis as a Service* aimed at solving such issues and we propose the first architectural blueprint implementing it. This novel concept, which can be considered an offspring of the more generic *Software as a Service (SaaS)*, has never been applied to the software (evolution) analysis before. The only existing similar approach, exploring services as a viable means for facilitating interoperability among tools, is the one explored by Jin and Cordy [JC05] with OASIS (Ontological Adaptive Service-Sharing Integration System). However, this work did not go past an initial small proof of concept.

Research Question. This work does not give a final answer to any specific research question. However, it lays the theoretical foundations to an approach that aims to eventually answer all four of them.

2. ***SOFAS, the Implementation of Software Analysis as a Service.***

Problem. How can we effectively implement the concept of *Software Analysis as a Service*?

Contribution. In this study, we present in detail the concrete architecture for the *Software Analysis as a Service* concept, its design considerations and implementation aspects, as well as a set of ready-to-use services based on real usage scenarios. The result is what we consider a feasible architecture for distributed analysis services based on the ideas sketched in the previous work and built upon few initial experimental implementations. The proposed architecture follows the principles of a REST [Fie00] and allows for a simple yet effective provisioning and use of software analyses based upon the principles of Representational State Transfer around resources on the web. It is made up by three main constituents: Software Analysis Web Services, a Software Analysis Broker, and Software Analysis Ontologies. Web services expose already existing analysis tools as standard RESTful web service interfaces. The Software Analysis Broker acts as the services manager and the interface between the services and the users. It contains a catalogue of all the registered analysis services with respect to a specific software analysis taxonomy. Software Analysis Ontologies define and represent the data consumed and produced by the different services. In this way, analyses are no longer bound to integrated development environments, but they are accessible on the web through a common web architecture.

This study also contains a first brief description of the ontologies we use to describe the data produced and consumed by *SOFAS*' services.

Evaluation. We validate the devised architecture with a use case scenario. In particular, we show how a combination of *SOFAS*' services can support a user in a concrete software quality analysis task: finding the code smells of the major releases of ArgoUML[5]. We also present other two concrete use cases that show *SOFAS*' versatility and usefulness. The first one is a Microsoft Surface application that uses the data produced a single service for purposes of multi-touch enabled code navigation and design recovery [MWS+12]. The second one, SMELL TAGGER [MFGW12], uses the data produced by a combination of services to detect and visualize the overall code structure, code smells [FBB+99], multiple evolution metrics using kiviat diagrams [PGFL05] and the house metaphor [BG07] of a software system for collaborative code reviews. For these tools and their evaluation, some of the most popular Java-based open source projects have been analyzed (*e.g.*,, ArgoUML, Eclipse, Vuze, jUnit, Tomcat, Derby). At last, some of *SOFAS*' services have been used extensively by external research groups to, for example, study factors of success and failure in open source projects or the contribution and collaboration patterns in OSS projects.

Research Question. With the full formalization of *SOFAS*' architecture we answer the research question: *How can the functionalities of this plethora of evolution analyses be offered in a consistent and efficient way?* (**RQ1**).

3. ***Ontologies, the Means to Describe the Product of Software Analysis Services.***

Problem. REST provides us a truly uniform interface to describe *SOFAS* services, the structure of their input and output and how to invoke them at a syntactic level. However, there is no way to programmatically know what a service actually offers and what the data it consumes/produces means.

Contribution. In this study, we critically reflect on the potential that the Semantic Web yields for *SOFAS*, and software evolution in general, to overcome the aforementioned problem. In particular, we show four characteristics that are most beneficial for the field: shared taxonomies, extensible meta-models, explicit relations, and Linked Data. Moreover, we present in detail *SEON*, our

[5] http://argouml.tigris.org/

family of software evolution ontologies (which we briefly introduced in the previous work). These ontologies describe knowledge on multiple levels of abstraction, ranging from code structures up to stakeholder activities.

Evaluation. We validate *SEON* by describing three semantics-aware tools that make extensive use of it to help developers in dealing with large amounts of software evolution data: *SOFAS*, HAWKSHAW a natural language query interface for developers and large-scale software visualization.

Research Question. With this work, we address the research question *How can the heterogeneous and wide-ranging data produced by such analyses be effectively described?* (**RQ2**).

4. ***Another Use of Evolution Ontologies in Software Engineering.***

Problem. How can the software evolution ontologies we created be exploited to help software engineers to answer program comprehension task in a different way?

Contribution. This work presents more in detail one of the previously introduced approaches that make use of *SEON*, namely the natural language query interface for developers, integrated into Eclipse. This system allows software engineers to use guided-input natural language strongly resembling plain English to query for information about a software system. As a first proof of concept to demonstrate the potential of such an approach, we focused only on supporting queries concerning static source code information, such as "How often is this field accessed?" or "What are the subclasses of this class?". This work is not strictly related to the main research theme of this thesis. As a matter of fact, it is a key part of the main body of work of Michael Würsch's PhD thesis, in which the author collaborated. It was included in this thesis to further prove and strengthen *SEON*'s usefulness and versatility claims of the previous study.

Evaluation. We evaluate the proposed approach with a case study in which we demonstrate–by the example of the open source library JFreeChart[6]–that it can be effectively used to answer the most common program comprehension questions that arise during software evolution tasks [dAM08].

[6] http://www.jfree.org/jfreechart/

Research Question. With this work, we strengthen the answer to the research question *How can the heterogeneous and wide-ranging data produced by such analyses be effectively described?* (**RQ2**) that we provided in the previous work.

5. ***A Composition Framework Built on Top of SOFAS.***

Problem. *SOFAS*' services have already been proved effective and useful. However, how can we successfully compose them into complex, high level, workflows? What are the benefits of such composition on software evolution analysis?

Contribution. This study presents a novel framework for semi-automated software analysis composition that we integrated into *SOFAS*. It explains how such composition works and describe *SCoLa*, a new language we devised to define the composition of analyses and model workflows. This framework exploits the RESTful nature of *SOFAS* and comes with a service composer to enable semi-automated service compositions by a user. We also present in detail two different approaches using such workflows to support different stakeholders in gaining a deeper insight into a project history and evolution. Both cases show the composition of many different types of analysis into a workflow, but with different purposes.

Evaluation. We use the two approaches presented in this work for a proof of concept validation. The first application conceptually proves that our framework can be used to address relevant evolution analysis questions, such as, finding code locations (i.e. hotspots) that have a high change frequency, intensive change coupling with other entities, and exhibit code clones. The second application shows how tools can harness such workflows to automatically gather a wide range of varied yet interlinked information about a software system and how they can use that for their own specific needs. Both approaches were used during a quality assessment process of a commercial software we carried out with an industrial partner. As an additional use case validation, they have been also used to analyze some of the most popular Apache Commons[7] projects.

Research Questions. With the proposed composition framework we address the research question: *How can these analysis be (semi)-automatically composed to provide more complex higher-level analyses?* (**RQ3**). With the two concrete

[7] http://commons.apache.org

workflow applications we provide a first answer to the research question *What is the impact of our proposed platform on the field of software evolution analysis?* **(RQ 4)**.

6. ***Using SOFAS to Replicate Mining Experiments.***

   Problem. Does our approach impact software evolution analysis in any other way? In particular, can we use *SOFAS* to facilitate and support the replication of software evolution empirical studies?

   Contribution. In this study, we empirically evaluate the potential of *SOFAS* in replicating empirical studies on software evolution published throughout the years at the Working Conference on Mining Software Repositories (MSR[8]). We show that we can replicate, to different degrees of completeness, up to 62% of these studies. Studies that can be fully replicated account for 30% of the total, while the remaining 32% are studies that can only be partially replicated. A study is considered partially supported if its results cannot be replicated out of the box, but the ground data from which they are derived can be calculated.

   Evaluation. To corroborate the replicability claims reported in the study and to better show the potential of *SOFAS* in such a replication context, we replicate one of such studies and present in detail how the replication was carried out and the final results.

   Research Question. With this study we provide a second answer to the question *What is the potential impact of our proposed platform on the field of software evolution analysis?* **(RQ 4)**. In particular, we prove the impact on the replicability of software evolution empirical studies.

7. ***A Possible Research Offshoot.***

   Problem. Currently existing version control system are the major bottleneck and single point of failure in any analysis that uses version history. This is because such systems are not built to be systematically analyzed. The only way to retrieve the historical data they contain is through the parsing and analysis of the bare history log; which is far from being the optimal approach. Because of that, incremental, proactive processing is barely supported and retro-active computations are long, resource-intensive and often error prone. In order

[8] www.msrconf.org

for software evolution analysis to play a part in the developers' day-to-day processes and to prove its immediate usefulness, a new type of version control system should be devised.

Contribution. This work represents one of the several future research directions inspired by this thesis. We propose an architectural blueprint for a plug-in based version control system in which analyses can be directly plugged into it in a flexible and lightweight way. Analyses can thus register for specific repository events (e.g., a commit, the tagging of a new release, etc.) and, based on them, automatically and proactively run and update their data every time it is needed in an incremental fashion. Moreover, with the data produced, they can also enrich the limited VCS data already existing (e.g. with source code metrics, fine grained source code change information, etc.). With our proposed approach evolution analyses can finally blend into version control systems in a transparent and lightweight way, without having to rely on the analysis of history logs.

Evaluation. No actual evaluation of the proposed approach has been carried out yet. However, we developed a first proof of concept prototype, featuring a stripped down version of all the presented architectural components and of two initial analysis plug-ins.

1.5 Validation

Software frameworks are universal, reusable software platforms used to develop applications, products and solutions. They are a set of code or libraries that provide functionalities common to a whole class of applications. While one library usually provides one specific piece of functionality, frameworks offer a broader range of them. Thus, rather than rewriting commonly used logic, a programmer can leverage a framework, which provides often used functionality, limiting the time required to build an application and taking full advantage of any preexisting knowledge. Furthermore, a framework can be extended by its users to add new functionalities while its core functionalities remain immutable.

In our work, we focus on the evaluation of *SOFAS* along three different dimensions that are inspired by these core properties and that are key to any framework:

Applicability. How the framework can be effectively exploited by other tools and applications and what its benefits are. In our case, we demonstrate how *SOFAS* has already been used in several concrete analysis scenarios to extract and calculate wide-ranging analysis data for over then 300 software projects (mostly open source).

Flexibility. The breadth of uses of the framework and its adaptability to different scenarios. The same use cases used in evaluating *SOFAS*' applicability also prove its flexibility. In fact, they use data produced by *SOFAS* and compose its services for a wide range of real-world uses, from software visualization to software quality assessment and program comprehension. This is mainly due to our framework's support for custom analysis composition. This feature allows users to freely compose analyses into user-defined and very diverse workflows, making use of *SOFAS* in often previously unforeseen ways

Relevance. The impact of *SOFAS* on software evolution analysis and software evolution in general. This is addressed throughout the thesis, as for every work or component we present and evaluate, we also elaborate on its relevance in the field. However, it is the central point of the replication of mining experiments presented in Chapter 7, in which we show that up to 62% percent of MSR empirical studies can be replicated (to different degrees of completeness).

Due to the emphasis on these three dimensions, we mainly employ use cases rather than, for example, a comparative study. In our opinion, they suit our needs better than any other technique, in particular in evaluating the applicability and flexibility of *SOFAS*. With a

combination of them we are able to show how our solution can be used for several, diverse software analysis purposes and to study wide corpus of systems (over 300). Furthermore, our goal is not to compare our approach with other similar solutions to show whether we outperform them or which features we support and which one we do not. In our opinion, such a study is more appropriate for the comparison of tools or algorithms. On the other hand, the main goal of this evaluation is to prove by induction that *SOFAS* and its combination of Semantic Web technologies and REST offers a viable solution to the problem stated in the thesis motivation.

For the sake of clarity, we split this evaluation between the three major milestones of our approach: (1) the concrete implementation of concept of *Software Analysis as a Service* with *SOFAS*, (2) the ontologies (*SEON*) we use to describe the produce of the software analysis service or (3) the analysis composition framework we built on top of *SOFAS*.

Evaluation of *SOFAS* and the Software Analysis as a Service concept

We show and validate the applicability, flexibility and effectiveness of *SOFAS* with three use cases.

In the first one, we show how a combination of *SOFAS*' services can be used in a concrete software quality analysis task: finding the code smells of the major releases of ArgoUML. We describe which *SOFAS*' analyses would be needed and how they should be arranged together into a workflow that eventually produces the data to answer that question. In the other two use cases, we focus more on showing *SOFAS*' flexibility and concrete applicability as it is being used by two existing tools developed by other researchers.

In the first use case, a Microsoft Surface (now called PixelSense[9]) application uses the data produced by a single service for purposes of multi-touch enabled code navigation and design recovery [MWS+12]. In the second one, SMELL TAGGER [MFGW12], a collaborative code review application, uses the data produced by a combination of services to detect and visualize, on a multitouch screen, the overall code structure, code smells [FBB+99] and multiple evolution metrics using different visualization paradigms (*e.g.*,, kiviat diagrams [PGFL05] and the house metaphor [BG07]). For the evaluation of both tools, the authors analyzed some of the most popular Java-based open source projects, *e.g.*,, ArgoUML, Eclipse, Vuze, jUnit and the entire Apache Software Foundation codebase (more than 300 projects, including Tomcat, Derby, Subversion, Apache HTTPD, etc.). This demonstrates the ability of *SOFAS* to support users (humans and computers) in the

[9] http://www.microsoft.com/en-us/ pixelsense

extraction and computation of valuable information from software repositories.

At last, some of *SOFAS*' services have also been used by external research groups to, for example, study factors of success and failure in open source projects or the contribution and collaboration patterns in OSS projects. Furthermore, these groups come from very different backgrounds, such as Physics, Management and Economics, which shows how with our approach we can open the door to software evolution analysis to researchers other than software engineers.

Evaluation of *SEON* and its Use in Software Engineering

SEON provides a shared taxonomy of important software engineering concepts that has already found multiple applications. For this reason, we exploit these applications to validate such ontologies and prove their usefulness and flexibility in supporting different software engineering tasks. First, we showcase three existing semantics-aware tools that make extensive use of *SEON* to help developers deal with large amounts of software evolution data. Second, we further evaluate one of these tools with a concrete use case.

For the first validation case, we the discuss in detail the three different applications that work with *SEON* as their semantic backbone. The first one is *SOFAS*, the main focus of this thesis (which, as reported in this chapter, has been evaluated in several other ways). The second one is HAWKSHAW, a natural language interface for answering program comprehension questions. The third application is a recommender system called SEMANTIC VISUALIZATION BROKER (SVB). SVB analyzes the semantics of a given set of data and comes up with a list of visualizations that could be helpful to gain a deeper understanding of the software system under analysis. These three approaches have been fully implemented in proof-of-concept tools and they show the wide range of the possible uses of *SEON*. For this validation, in the case of *SOFAS*, we show how it exploits *SEON* as a formal description of the input and output of its individual services. With HAWKSHAW, we show how the clear semantics of OWL can be be exploited to translate program comprehension questions formulated by developers in quasi-natural language. At last, with the SVB, we illustrate how, by exploiting the Semantic Web's reasoning and explicit relations, we can automatically infer suitable visualizations for given sets of data and present them to users. All of these three applications would have been significantly harder to implement without *SEON* and the use of Semantic Web technologies.

In the second validation of *SEON*, we use a case study to demonstrate, by the example of the open source library JFreeChart[10], that, together with HAWKSHAW, it can be effec-

[10] http://www.jfree.org/jfreechart/

tively used to answer the most common program comprehension questions arising during software evolution tasks [dAM08]. In particular, we show that, compared to existing tools, developers are given more flexibility when composing questions with our approach as they can formulate them conveniently using different variations of natural language sentences. Michael Würsch's thesis provides an additional use case validation of HAWKSHAW.

Evaluation of the Composition Framework Built on Top of *SOFAS*

The analysis composition framework we built on top of *SOFAS* can be considered the final concrete outcome of this thesis, bringing together everything that was previously laid out. Therefore, the goal of this last validation is bifold: to evaluate the composition framework itself and to perform a final, more comprehensive evaluation of the relevance of *SOFAS* as a whole. It comprises of three use cases. The first two focus on showing the impact of *SOFAS* on gaining a better understanding of the quality of a software system and its evolution. The third aims at proving *SOFAS*' versatility by showing how it can be successfully used to replicate empirical studies on software evolution.

In the first use case we show in detail how we can answer the question "Which are the hotspots and evolution anomalies for a project?" by composing a specific workflow using the proposed framework, its web UI and its custom composition language *SCoLa*. This question and the associated workflow originate from a concrete need we encountered while performing a software quality audit of a commercial software. With such a use case, we conceptually prove that our framework can be used to address relevant evolution analysis questions.

Analysis workflows themselves are extremely valuable in answering specific evolution analysis questions and in singling out, unequivocally, noteworthy entities. However, when used by themselves, they lack the capability to fulfill broader and more open-ended information needs. For example, giving an overall view of the evolution or the current state of a software project or to show trends of specific, critical metrics. They can still provide all the information needed to fulfill those needs but, in this case, human interpretation is heavily needed to put everything into context and draw meaningful conclusions. The second use case illustrates how tools, in this case our web application Software Evolution Perspectives, can harness *SCoLa* workflows to exactly fill such a gap. In particular, we demonstrate how it is able to automatically gather a wide range of varied yet interlinked information about a software system and how it uses that to give users a detailed and intuitive overview on the quality of a software project and its history. This is achieved through the use and combination of different "perspectives" focusing on different aspects

of the software analyzed. Every perspective offers different interactive visualizations of the aspect addressed, along with automatically generated considerations about it, for users to better grasp the implications of the data being shown. Also in this case, this application has been used for a software quality audit of a commercial software during a collaboration with an industrial partner. As an additional use case validation, both approaches have been also used to analyze some of the most popular Apache Commons[11] projects.

In the third use case, we prove how *SOFAS* can be used as a platform to support the replication of empirical studies on software evolution (a long standing issue in the field) and, to a certain extent, more generic software analyses. In particular, we show that we can replicate, to different degrees of completeness, up to 62% of all the empirical studies published at the Working Conference on Mining Software Repositories (MSR) from 2004 to 2011. In order to do that, we first perform a complete literature review of all the works published in the proceedings of all MSR conferences up to 2011 to retrieve all the paper that present empirical studies (51% of the total). After a broad categorization of the studies found, we then inspect in detail each one of them to assess if and how it could be replicated using SOFAS. Studies that can be fully replicated account for 30% of the total, while the remaining 32% are studies that can only be partially replicated. A study is considered fully replicable only if the same results can be replicated or if all the necessary data can be calculated with the exception of its final aggregation or interpretation. On the other hand, a study is partially replicable if its results cannot be replicated out of the box, but the ground data from which they are derived can be calculated. To then further corroborate such replicability claims–and to better show the potential of *SOFAS* in such a replication context–we replicate one of the studies found in the previous survey, show in detail how the replication was carried out and present the final results.

1.6 Thesis Roadmap

The remainder of this thesis is composed of Chapters 2-9. Chapters 2-8 are based on the studies that frame the foundation of the thesis (see Section 1.4).

In Chapter 2 (33 et seq.) we introduce the founding concept of *Software Analysis as a Service* aimed at solving the core research issues motivating the thesis (see Section 1.3) and we propose the first architectural blueprint implementing it. With then present the full-blown, complete working architecture in Chapter 3 (53 et seq.). In Chapter 4 (75 et

[11] http://commons.apache.org

seq.) we present in details *SEON*, our family of software evolution ontologies used to describe the data consumed and produced by *SOFAS* services. We also present two other semantics-aware tools making extensive use of *SEON*. One of these, a natural language query interface for developers integrated into Eclipse, is then described in more details in Chapter 5 (109 et seq.). Even though this approach is not strictly related to the main topic of this thesis, we included it to show another use of *SEON* and further prove its usefulness and versatility. Chapter 6 (135 et seq.) presents a framework for semi-automated software analysis composition we integrated into *SOFAS*. Two different use cases are then presented to show how, with this framework, different analyses can be composed into workflows addressing relevant evolution analysis questions. In Chapter 7 (173 et seq.) we then further evaluate the relevance of our approach by showing the effectiveness of *SOFAS* in replicating the empirical studies on software evolution published at the Working Conference on Mining Software Repositories (MSR). In Chapter 8 we then elaborate on a possible research offshoot inspired by the experience we gained while working with *SOFAS*. Figure 1.2 shows the roadmap for the remainder of this dissertation. It lists each of the remaining chapters and the details of the corresponding publications.

Chapter 9 (215 et seq.) concludes the thesis, discusses the implications of our findings, and highlights possibilities for future work.

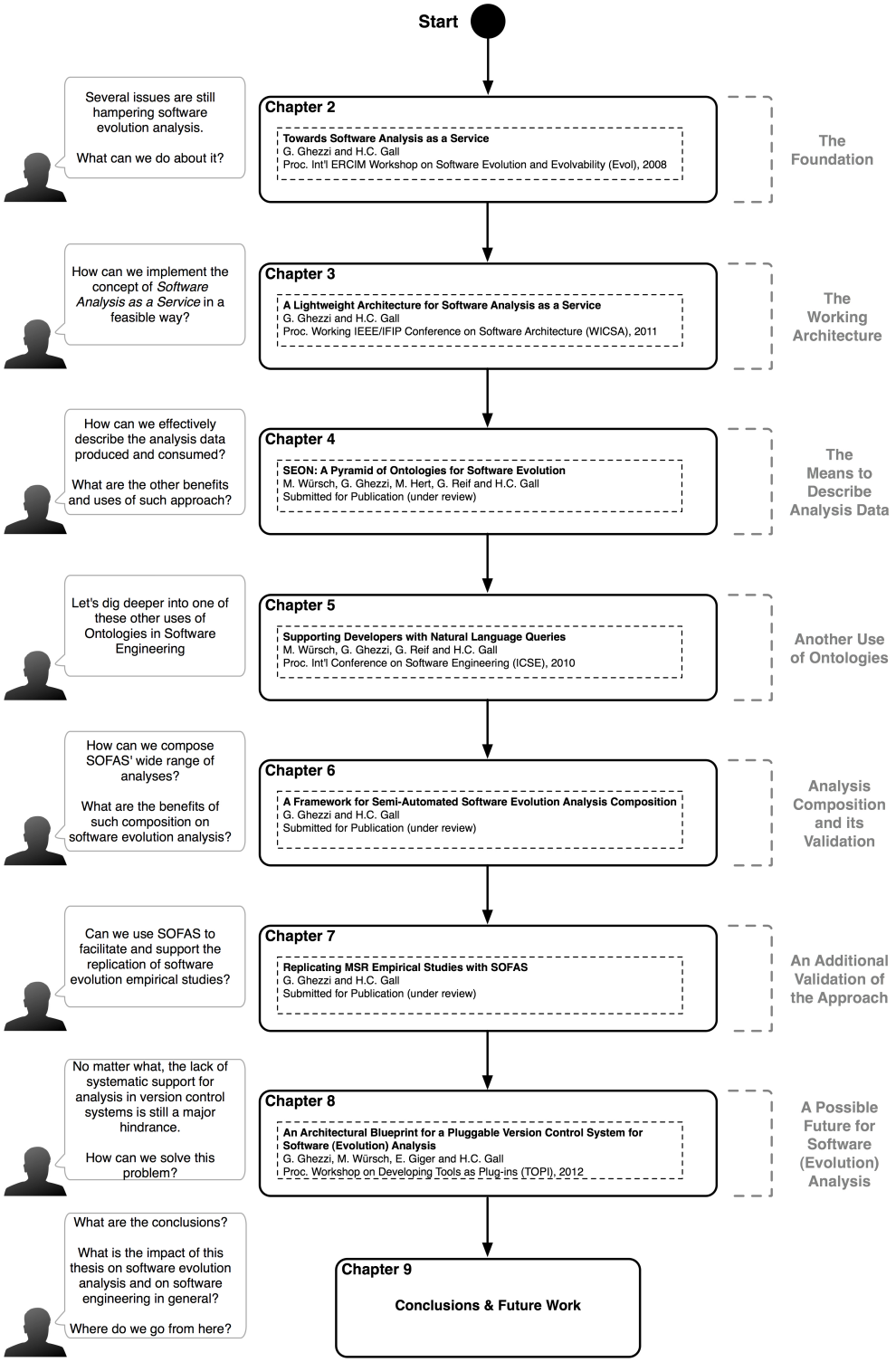

Figure 1.2: The thesis roadmap, showing the relation between the key works forming the thesis, the individual chapters and their associated publications.

2

Towards Software Analysis as a Service

Towards Software Analysis as a Service
Giacomo Ghezzi and Harald C. Gall
Proc. Int'l ERCIM Workshop on Software Evolution and Evolvability (Evol), 2008
DOI: 10.1109/ASEW.2008.4686315

THROUGHOUT the years software engineers have come up with a myriad of specialized tools and techniques that focus on a certain type of analysis, such as metrics extraction, evolution tracking, co-change detection, bug prediction, all the way up to social network analysis of team dynamics. However, easy and straight forward synergies between these analyses/tools rarely exist because of their stand-alone nature, their platform dependence, their different input and output formats and the variety of systems to analyze. This significantly hampers their usage and reduces their acceptance by other researchers and software companies. To overcome this problem we propose a distributed and collaborative software analysis platform to enable a seamless interoperability of software analysis tools across platform, geographical and organizational boundaries. In particular, we devise software analysis tools as services that can be accessed and composed over the Internet. These distributed services shall be widely accessible through a software analysis broker where organizations and research groups can

register and share their tools. To enable (semi)-automatic use and composition of these tools, they are classified and mapped into a software analysis taxonomy and adhere to specific meta-models and ontologies for their category of analysis.

2.1 Introduction

Successful software systems must change or they become progressively less useful, but as they evolve, they become more complex and consequently more resources are needed to preserve and simplify their structure [LB85]. Studies estimate the costs for the maintenance and evolution of large, complex software systems from 50% to 95% of the total costs in the software life-cycle. To reduce these costs several techniques and tools have been developed: to discover components that need to be modified when a new feature is integrated, to detect architectural shortcomings, to detect error prone modules, or for project managers to estimate the maintenance costs and allow for better planning, etc.

Software analysis tools mainly focus on a particular kind of analysis to produce the results wanted by the engineer. So, if different analyses are required, the engineer needs to run several tools, each one specialized on a particular aspect, ranging from pure source code analysis to duplication analysis, co-change analysis and visualization. In addition to this, all these techniques have their own explicit or implicit meta-model which dictates how to represent the input and the output data. Thus the sharing of information between tools is at most possible by means of a cumbersome export towards files complying to a specified exchange format. Also, if there exist services of the same kind (e.g. duplication analysis) there is no way of comparing the results or integrating them other than manual investigation. And there are even more issues that limit tool interoperability as for example, platform and language dependence.

We claim that this status quo severely hampers software evolution research and a critical assessment of the research fields uncovers the fact that people keep re-inventing the same wheels with little advancement of the field as a whole.

So our goal is *to devise a distributed and collaborative software analysis platform to allow for interoperability of software analysis tools across platform, geographical and organizational boundaries*. Such tools will be mapped into a software analysis taxonomy and will adhere to specific meta-models and ontologies for their category of analysis and offer a common service interface that enables their composite use on the Internet. These distributed analysis services will be accessible through an incrementally augmented software analysis catalog, where organizations and research groups can register and share their tools.

The remainder of this paper is organized as follows: Section 2.2 gives an overview about current software analyses to be supported by a service platform. Section 2.3 explains our proposed approach, going over its main constituents. In Section 2.4 we go over the first

prototype we implemented upon which all the future work will be based on. In Section 2.5 we outline a first use case scenario we intend to use as a first proof of concept in order to validate our ideas. The works related are presented in Section 2.6. We conclude in Section 2.7 with a discussion and future work which will be built upon from the ideas presented in the paper.

2.2 Software analyses to be supported

Software analysis is one of the key activities in software engineering as it allows to extract the most diverse and extensive information regarding a software system. The classic analyses have been for years the ones targeting models and source code [JR00]. In the last years many research groups have shifted their attention to software evolution and the whole established community of reverse engineering, reengineering, and program understanding has actually acknowledged that evolution is indeed the umbrella of their research activities.

Software analysis research can be divided in three main categories, with regard to what topic they address:

- Development: the extraction and/or the analysis of information about the development of software artifacts.
- Models: the extraction and/or the analysis of models representing different features and views of software artifacts.
- Code: the analysis of software artifacts' source code to extract information and assess properties as well-formedness quality, correctness, etc.

There is a plethora of research on these topics, but it is not in our intention to give a complete picture of the state of the art. We just want to give a brief overview of the current analysis techniques to setup a service platform for software analyses.

Approaches focusing on the software development either study its source code change history, bug history, its underlying dynamics or a combination of them. Fischer et al. [FPG03b] populated a release history database, combining information from version control and bug tracking systems, namely CVS and Bugzilla to facilitate further analysis. Draheim et al. [DP03] had a similar approach but only worked with version control data from CVS. Many other works detect and track changes made on the source code during the software project lifetime. Zou et al. [ZG03] used origin analysis to detect merging and

splitting while S. Kim et al. [KPW05] used it to track function name changes. M. Kim et al. [KSNM05] focused just on code clone evolution and built a clone genealogy tool to extract code clones history from a project CVS repository.

Works by Zimmermann et al. [ZWDZ04] and Ying et al. [YMNCC04] tried to predict future source code changes given past source revision history of a project stored into CVS repositories to then recommend potentially relevant source code for a particular modification task.

Source revision history is analyzed to extract also other kinds of information. Livshits et al. [LZ05] combine that with dynamic analysis techniques to identify application-specific patterns and find pattern violation. Hipikat [uM03] forms an implicit group memory combining CVS source repository data, Bugzilla data, messages posted on developer forums and other project documents to recommend artifacts that are relevant to a particular task that a developer is trying to perform.

Gall et al. [GJK03] extracted logical couplings of software modules by analyzing CVS data, in particular check in and check out time and the authors of those actions, and from that they were able to discover design flaws without analyzing a single line of code. Fluri et al. [FG06] focused on the extraction of several fine-grained source code change types and the assessment of their significance in terms of their impact on other source code entities and whether a they may be functionality-modifying or functionality-preserving. Then, Nagappan et al. [NB05] predicted defects density for a system using code churn metrics fetched from its change history.

Similarly to the works on source code change, bug analysis addressed extraction of data from a bug repository (as we already saw in [FPG03b]), its prediction or its analysis. For that, Hassan et al. [HH05] developed a dynamic cache of the ten mostly error prone subsystems (directories). Kim et al. [KZJZ07] proposed a similar approach, but they dynamically cached the most likely fault prone source code locations. Sliwerski et al. [SZZ05a] related version history and a bug database to detect, as Kim et al. [KZJZ07], code locations whose changes had been risky in the past and annotated them with color bars to show their risk rate in Eclipse [dRW04]. While much effort has been spent on software cost/effort prediction, very little has been done on bug fixing effort prediction. As for example the work by Weiss et al. [WPZZ07] in which, for every new bug report in a issue tracking system, similar earlier reports are fetched and their average time is used as a prediction for the new one.

Not only the history of a software development process has been addressed, but also its underlying dynamics. In particular, a lot of research has also been performed on the

role of the developers in evolutionary processes. For example, Čubranić et al. [uM04] and Anvik et al. [AHM06] both developed approaches for bug triaging that recommend a set of developers with the appropriate expertise to solve a particular bug by applying machine learning techniques on bug reports fetched from a bug repository (in these cases Bugzilla). Mockus et al. [MH02] located people with desired expertise not using bug reports but by analyzing data from change management systems. Girba et al. [GKSD05] analyzed CVS logs to reconstruct code ownership to help in answering which authors are knowledgeable in which part of the system and also reveal behavioral patterns: when and how different developers interacted in which way and in which part of the system.

Most of the approaches focusing on models target the reverse engineering of a wide range of abstractions and forms of representations from software systems.

For example, some work has been done to address UML models. Kollman et al. [KG01] and Tonella et al. [TP03] extracted UML Collaboration Diagrams by statically analyzing the source code, while Rountev et al. [RC05] extracted UML Sequence Diagrams. Some other works studied the runtime behavior of a software to recreate UML State Machine Diagrams [LBL06, Sys00] or UML Sequence Diagrams [GZ05, LBL06]. There is then a score of UML modeling tools that allow reverse engineering of Class Diagrams, from open source ones as Fujaba[1](which offers also basic Activity Diagram reconstruction) and ArgoUML[2] up to commercial ones as Together[3] and IBM Rational Rose[4].

Most works target CVS repositories as there is a great deal of big and significant open source projects that use it (as CVS itself is opensource), thus giving researchers a huge amount of information that can be freely studied and analyzed, but as more and more projects are being now moved to SVN, we expect that also researchers will soon start to focus significantly also on it.

2.3 Our Approach

There is a huge variety of tools and techniques out there offering the most disparate analyses on a software system. Such analyses are currently offered on a purely local basis and have their own distinct input and output format making it impossible to combine and integrate them effectively. What follows is the description of how we tackle the problem.

[1] http://wwwcs.uni-paderborn.de/cs/fujaba/

[2] http://argouml.tigris.org/

[3] http://www.borland.com/us/products/together/

[4] http://www-306.ibm.com/software/awdtools/developer/rose/index.html

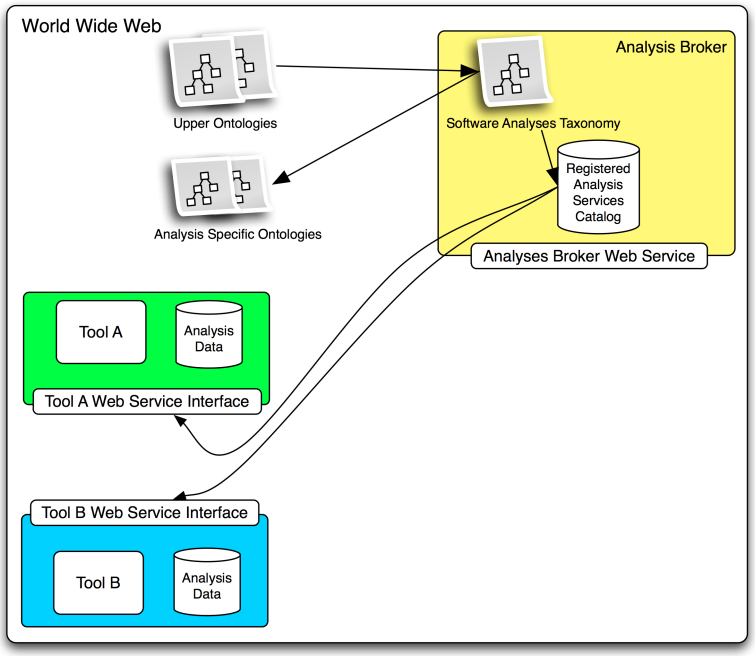

Figure 2.1: Overview of our software analysis service platform

Figure 2.1 gives an overview of our approach, which is made up by four main constituents: several software analysis web services, an analysis services catalog, an analysis broker and ontologies.

Software analysis web services are "wrappers" of already existing analysis tools exposing their functionalities and data through a web service.

The analyses catalog, as the name suggests, classifies all the registered analysis services with respect to a specific taxonomy we defined and stores other information about them.

The analyses broker acts as the interface between the catalog and the users.

Specific ontologies are used to define and represent the data consumed and produced by the different types of analysis, while upper ontologies define much more generic concepts common to several specific, ontologies, thus providing semantic links between them (otherwise they would remained decoupled).

In the following sections we explain in greater detail what these constituents are, what they do, and the benefits we can gain by using them.

2.3.1 Software Analyses as Web Services

Our solution proposes software analyses to be available as web services. We decided to leverage this paradigm as it is a well known standard and it was devised to overcome some of the problems we also face and thus already offer many of the features we need, namely: language, platform and location independence and service composition.

Independence is achieved with the use of XML-based language to describe the services (WSDL [CCMW01]) and a simple, lightweight communication protocol (SOAP [GHM+07]) intended for exchanging structured information, formatted into XML-based messages, in a decentralized, distributed environment, normally using HTTP/HTTPS, which allows also easier communication through proxies and firewalls. Composition and orchestration is provided by BPEL4WS (Business Process Execution Language for Web Services) [JE07a], an XML-based language designed to enable task-sharing for a distributed computing - even across multiple organizations - using a combination of Web services. Using BPEL, a programmer formally describes a business process that will take place across the Web in such a way that any cooperating entity can perform one or more steps in the process the same way.

To share a software analysis three things would need to be done:

- *Write and publish a web service* offering the methods to perform that particular analysis and to fetch the results, formatted with the ontology specific to that analysis.
- *Write an adapter* that calls the actual underlying tool, doing the necessary data format translations: from the web service input format (represented, as we explained, by a specific ontology) to the tool specific one and vice versa, from the tool output to the web service output, represented by the ontology defined for the specific analysis the tool is offering.
- *Register the service* on the analysis catalog to make it available to anyone interested.

As it can be seen, the internal logic, the input and output formats used, the platform and language under which the original tool runs will remain hidden behind the web service not being a burden for interoperability anymore.

More specifically, these analyses are the extraction and storage on our Release History Database [FPG03b] of CVS, Bugzilla data and FAMIX model of software projects offered by our EVOLIZER[5] platform and the ones offered by our CHANGEDISTILLER built on top

[5] http://seal.ifi.uzh.ch/evolizer/

of that, namely, the extraction of fine grained source code change [FWPG07], and their classification based on their significance level [FG06].

2.3.2 The Analyses Catalog

The Analyses Catalog, as the name suggests, stores and classifies all the registered analysis services so that they can be automatically discovered, invoked and their results fetched. To do that, a clear and univoque classification is essential, so, as a first step, we created a specific software analysis taxonomy to systematically classify the existing and future services.

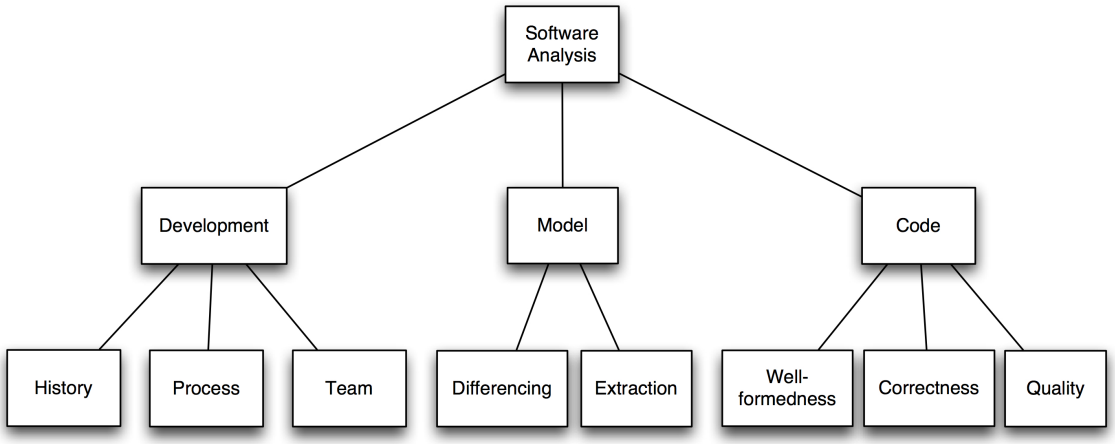

Figure 2.2: A condensed view of the taxonomy

Figure 2.2 gives a condensed view, due to space limitations, of that taxonomy. This taxonomy divides all the possible analyses in three main categories based on what their main focus is: the development of a software system, the underlying models of it and the actual source code. Each of those categories is in turn made up of many other subcategories.

Software development analyses are further divided into those targeting the history (extraction, prediction and analysis of source code changes and bugs), the process (its dynamics and metrics, as the ones defined by Lorenz et al. [LK94] and Nagappan et al. [NB05]) and the teams involved (their dynamics and metrics).

Model analyses are further divided into those targeting the extraction, either dynamic or static, of specific model representations (UML, FAMIX, call graphs, Rigi, etc.) and those computing differences between two models, usually of two versions of the same

system. Code analyses, being the oldest and thus most studied topics of this taxonomy, are by far the most numerous and thereby are further divided into many other categories, as for example those checking code well-formedness, its correctness and its quality.

We will not go into the details of this part for space limitations and because it is still a work in progress and the taxonomy is not yet stable and complete. The only part that is already stable is the one about code quality and in particular its subcategory containing analyses focusing on the design quality. We decided to first focus on this area as it is the most used in the field of software evolution analysis as it can show whether and how the quality of the system being studied evolved. Tools belonging to this category are, for example, the ones extracting and analyzing design metrics, as defined by Lanza et al. [LM05] and Lorenz et al. [LK94], and code smells, as defined by Fowler et al. [FBB+99], such as code clones detectors and predictors.

This proposed taxonomy is obviously not the only one possible and by no means complete, so it is most likely that some parts will be added on the way and some others will be modified. But the proposed categories are reasonable enough and make sense, in particular from the perspective of a user who wants to find some particular analyses without struggling with many and sometimes obscure categorizations but at the same time wants them to be expressive and meaningful. Since, to our knowledge, the literature lacks any preexisting taxonomies of this kind, we structured it manly using the currently existing approaches as a blueprint and so that they would "fit" reasonably well into that.

We chose to implement the whole taxonomy as an ontology, more precisely in OWL [SCM04], for three reasons.

1. We can achieve a formal representation of a set of concepts within a domain and the relationships between those concepts.

2. It is possible to reason about the properties of that domain and infer additional information based on the data explicitly provided.

3. Together with OWL we can use languages as SPARQL [PS08] to effectively query instances of the ontologies and fetch the services we are interested in.

In this way, the catalog is just an instance of an ontology, so it essentially comes down to an .owl file, which could then be published on the Internet to be accessed and queried by anyone who is interested, without the need of a web service to access it. But, since on top of that we wanted to offer other useful and more complex functionalities, we decided to make accessible through what we called the Analyses Broker.

2.3.3 The Analyses Broker

The Analyses Broker acts as a "layer" between the catalog and the users through which they can query, update, manage the catalog (namely register, update and unregister analysis services) and expand the taxonomy, as the one proposed is not supposed to be complete and new types of analyses that were not yet classified, or some modification to the already existing classification, could come up in the future.

Moreover, we can also offer more complex functionalities such as automatic composition of services. So, for example, if a user wants a series of analyses to be done on a project, the Broker would take care of finding, composing, executing them (for example with BPEL) and then just returning the final results to the user.

2.3.4 Ontologies

The large majority of the analysis tools is being used within institutional boundaries by single researchers who often are also the tool authors. The results of these tools are stored internally and are not accessible to third parties for the combination or integration of the results. Several researchers have pushed for common interchange formats such as GXL (Graph eXchange Language) [WKR01] or XMI [Xmi07], but their efforts have remained largely unheard. The MSR (Mining Software Repositories) community is striving for integration especially in their Mining Challenge track, but it is limited to the application of the analysis tools on the same case studies. The integration and combination of results, especially of different kinds of analyses, remain completely open and is the major challenge we need to tackle as its solution is one of the main motivations behind our work.

A promising alternative to solve those problems is to use ontologies, in particular OWL, to represent both results and input. With an ontology we define and enforce how the results (their structure and internal relation) of analyses belonging to a certain type would be. So any new service would have to support inputs and provide results conforming to the ontology defined for the specific analysis it offers. Furthermore, it gives us a sound and well known data format to use and the ability to share that data between different types of computers using different types of operating system and application languages, as it is written in XML.

But what really makes OWL stand out and worth using are the properties related to its ontological nature: (1) heterogenous domain ontologies can be semantically "linked" to each other by means of one or more upper ontologies, which describe general concepts

across a wide range of domains. In this way it is possible to reach some semantic interoperability between a large number of ontologies accessible "under" some upper ontology; (2) with OWL Description Logic foundation it is possible to perform automatic reasoning and derive additional knowledge; (3) we can use a powerful query language such as SPARQL or its extension iSPARQL [KBS07], that uses similarity operators to query for similar entities; and (4) in contrast to XML and XQuery [BCF+07] that operate on the structure of the data, OWL treats data based on its semantics. This allows for an extension of the data model with no backwards compatibility problems with existing tools.

2.4 The First Prototype

Having described the infrastructure, we will now show the main features of the first prototype of the Analysis Broker that we have developed. We decided to develop the Analysis Broker at first as it constitutes the foundation upon which everything else will be based on. In fact it is where all the future analysis services will be registered and through which a user or a tool would find and fetch services of interest.

More precisely, the Broker can be queried to get the content of the analyses catalog (in other words, the registered analyses) and if one or more specific analyses have been performed on some projects. We decided to offer just these two functionalities because in our opinion those two pieces of information are everything a user might want to know in this context, furthermore any additional information can then be fetched from a combination of them.

Those two queries are offered through a web service interface and the results formatted into a standardized machine readable format, more precisely OWL. In this way tools of any kind can (semi)-automatically fetch the analyses they need to then call them. However, this makes the results hardly readable by humans. So we chose to let the Broker be queried in the same way also through a website, which will format and present the results in a much more human understandable form. Therefore also here we will show the Broker functionalities through its website interface.

Figure 2.3 shows the initial view that will be presented to the user.

The user can do two things, either navigate through the catalog or query it. With the navigation option he/she can get an idea on the analysis taxonomy structure or see what the analyses being offered are. With queries, more specific information for the successive invocation of the services can be gathered.

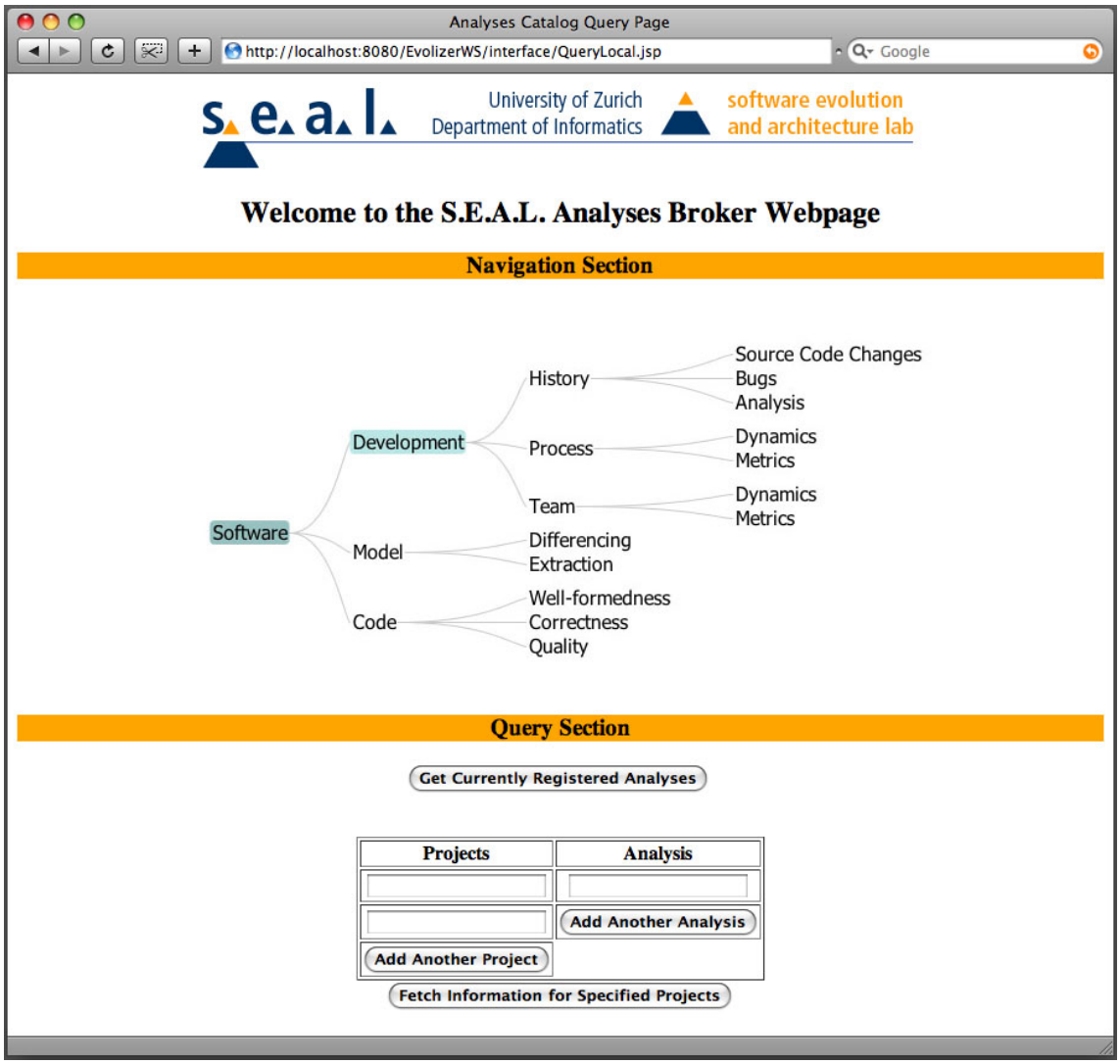

Figure 2.3: The initial page of the Analyses Broker website

Figure 2.4 shows what the Broker returns when queried for the currently registered analyses which is essentially the current instance of the catalog. So for every service is reported the name, the address through which it can be invoked and the type of analysis offered. Knowing the latter gives the user all the information on the service input and output. In fact, as we explained in Section 2.3.4, every analysis type is associated with ontologies to which the input and output of every service offering that analysis must conform.

Thus with this query it is possible to know what analyses can be performed and gather

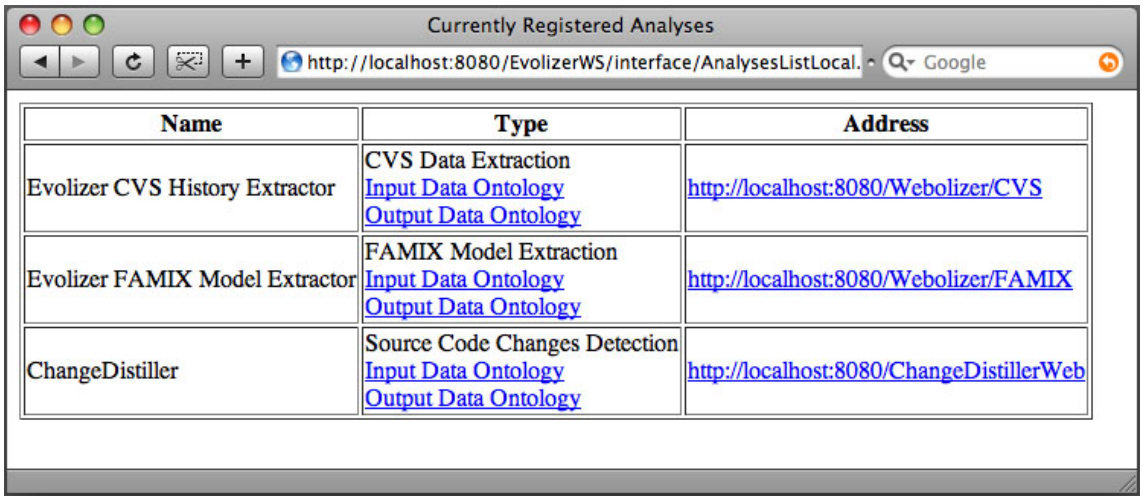

Figure 2.4: The registered analysis services

all the information needed to then call the ones that are of interest. So it will be used when a user or a tool, given a project, wants to conduct some analysis.

Figure 2.5 shows what the Broker returns when queried to get if one or more types of analysis were performed on for some specified projects. Note that for all the projects is displayed whether or not every single requested analysis has been already performed, without explicitly showing what is the actual service that did it.

In fact, as long as it is performed, it does not really matter who performed the analysis since, as we explained before, all the services offering it will comply to a common input and output. Nevertheless the address of the actual service offering the analysis is simply hidden by the html representation behind the "check" symbol. So it can be immediately invoked to get the available data without having to query the Broker for any other information.

All this provided information is useful to see what data about a project is already available to then fetch it or trigger the analysis to produce it. Furthermore, it can be handy for tools and users that need case study data from existing projects to then run their own analysis. For example a tool extracting some newly defined software project metrics might need CVS history data of software projects for case studies and proofs of concept for validation. So, instead of finding suitable projects and extracting their CVS data by itself, it could take advantage of the previous analyses and thus just fetch the data that has already been extracted by the registered services offering CVS data extraction.

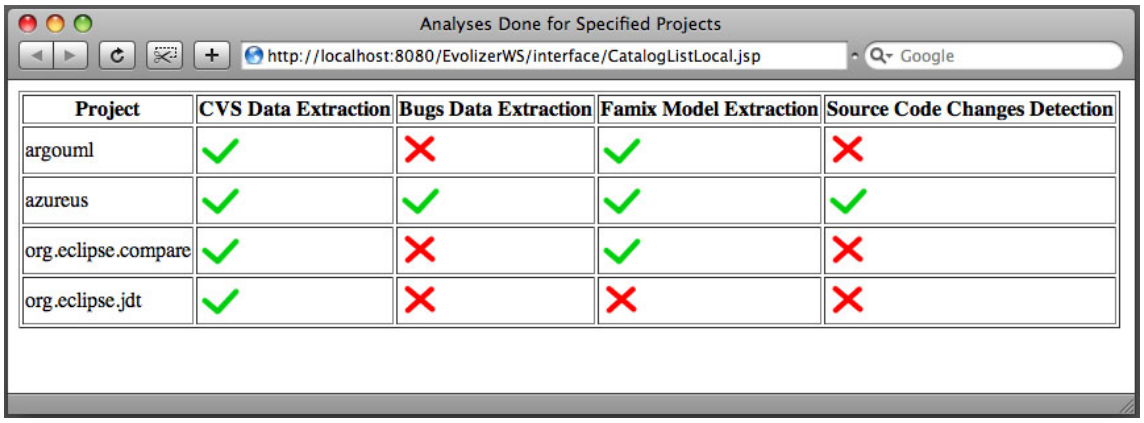

Figure 2.5: Broker list of analyses and projects

2.5 Validation

After having finished developing our prototype, we need to study and prove the potential of the Analysis Broker, the semantically enhanced catalog of software analysis tools and the usage of ontologies to represent and share analysis data.

Figure 2.6 sketches the scenario we have built for that first proof of concept, showing our solution at work.

We have created a test tool that plays the role of a development process analyzer which, given CVS versioning history data described by a CVS history ontology, extracts team metrics such as: the number of developers involved in each file, the total number of commits for each developer, etc. After that it returns the results to the user and stores them for future use by other tools.

In order to do that, our tool first queries the Analysis Broker to see if CVS history data of the chosen project has already been extracted by some service. If not, it will query again the Analysis Broker to get a list of the registered CVS history data extraction services and then ask one of them to carry out the extraction.

On the other hand, if the project history has already been extracted, it gets that data and performs standard predefined OWL queries and reasoning to get the information it needs to extract the wanted metrics. In fact, since all the services offering CVS history data have to represent and format the data they provide with a standard ontology it does not matter where the data comes from.

Once the metrics are computed, they are returned to the user and stored so that if the

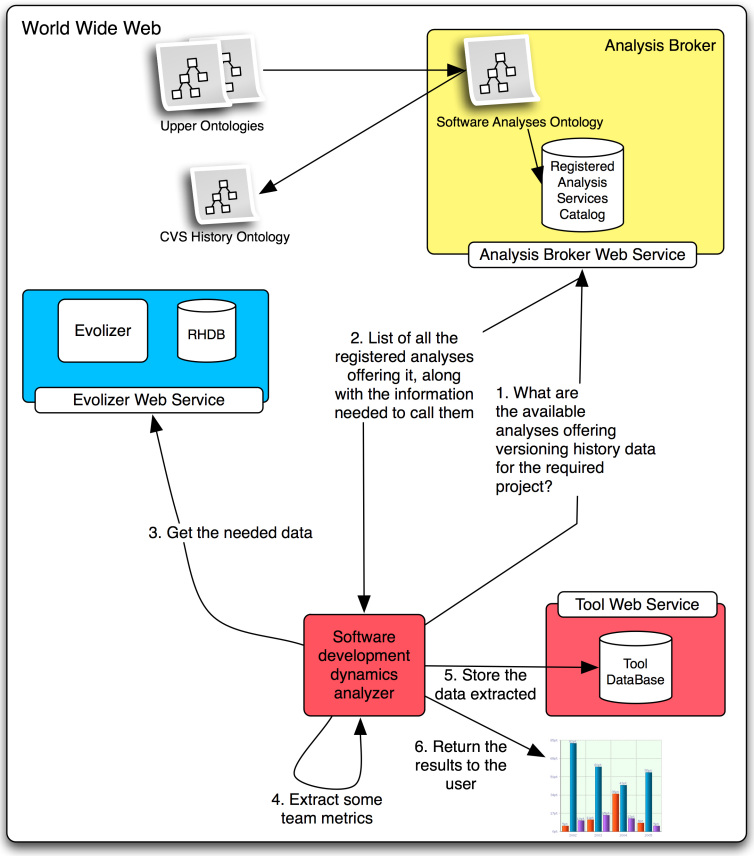

Figure 2.6: A first proof of concept

analysis was registered in the catalog and a proper web service to access it exist, other tools or people can retrieve that data.

2.6 Related Work

The use of web services and ontologies in connection with software analysis and evolution has, to the best of our knowledge, been addressed only recently by only a few researches.

A few works have addressed software analysis data and concepts representation with ontologies. Hyland-Wood et al. [HWCK06] presented an OWL ontology of software engineering concepts (SEC), including classes, tests, metrics and requirements. Happel et al. [HKST06] in their KOntoR approach stored and queried meta-data about software artifacts to foster software reuse. What is interesting for us is that they proposed various

ontologies to provide background knowledge about software components, such as the programming language and licensing models. Both works could be really valuable for us as upper ontologies, since they could provide us general concepts common to many specific software analyses ontologies, so that they could be semantically "glued" together.

Highly related to our approach is the work by Kiefer et al. [KBT07], which proposed EvoOnt, a software repository data exchange format including software, release and bug related information based on OWL. To effectively mine software systems represented in that OWL format and find, for example, code smells, they introduced iSPARQL, a query engine supporting similarity joins. From their work we borrow the idea of using ontologies to represent software analysis data to facilitate data exchange and automatic reasoning.

To the best of our knowledge Jin and Cordy [JC05], with their Ontological Adaptive Service-Sharing Integration System (OASIS), are the first and only researchers that so far studied an ontology based software analysis tool integration system that employs a domain ontology and specially constructed external tool adapters. They also implemented a proof of concept with three diverse reverse engineering tools that allowed them to explore service-sharing as a viable means for facilitating interoperability among tools.

We share with them the overall concept, but the two approaches have many differences as they have partially different goals. In fact, the objective of their integration effort was to allow the functionality/analysis available in one tool to be applied to the fact-base of another one. For this reason, they used an ontology just to describe the set of representational concepts that the different tools to be integrated might require and/or support. On the other hand, as we already showed, we intend to exploit ontologies on a much broader scale: to catalog and describe the services, to represent and standardize their input and output accordingly to the type of analysis offered, to semantically link different results and to perform (semi)-automatic reasoning on them.

Moreover, to overcome language, platform and location dependencies, we expose the functionalities of the different tools through web services, while they use not better specified ad-hoc adapters.

2.7 Conclusions and future work

The combination and integration of different software analysis tools is a challenging problem an engineer faces when he/she needs to gain a deeper insight on a software system and its history. For every required analysis a specialized tool, with its own explicit or

implicit meta-model dictating how to represent the input and the output data, has to be locally installed, configured and executed. Even if different analyses of the same kind exist, the only way to compare them is to do it manually.

In our opinion the combination of ontologies and web services we presented in our approach can be extremely valuable to solve that problem. Using web services to expose the functionalities offered by the analysis tools gives us total independence in terms of platform, language and location and the possibility in the future to explore the use of well known mechanism of composition and orchestration (e.g. BPEL4WS) of several analysis services. OWL ontologies specific to distinct types of analyses allow us to have standard formats to define and represent the data consumed and produced by the analysis services, which can then be integrated with each other thanks to semantic "links" provided by generic, upper ontologies. In addition to that, thanks to OWL's powerful query language (SPARQL) and its Description Logic foundation, data can be extracted and additional knowledge can be inferred with existing tools.

The purpose of this paper was to provide the foundations upon which subsequent improvements and implementations will be based on. With our use case we previously introduced, we will first validate and prove the potential of all features of our first prototype of the Analysis Broker: the semantically enriched catalog of software analysis tools and the usage of ontologies to represent and share analysis data. This phase will also help us to find weaknesses, refine and stabilize our approach and the prototype.

This is crucial because from there want to add other analyses to our catalog (possibly coming from other research groups), such as clone detectors and change predictors, to asses the feasibility and usefulness of the integration of results coming from different analyses. Furthermore, to make that possible we will also create and add upper generic ontologies that will semantically "glue" together the specific ones defining the tools output. We plan to re-use some of the existing ideas, as for example the software engineering concepts (SEC) ontology [HWCK06].

These are the first two main goals we aim at, as we hope that having a sound solution of proven usefulness would push other research groups to share their analysis approaches through our platform, making it really valuable. However, there are also many other ideas we want to explore in the future. Such as automatic recommendation of services given specific user analysis needs and automatic composition and orchestration of services so that a user can choose a sequence of analyses he wants to be performed on a software system, have it automatically executed and be presented with the final results.

Finally, we are convinced that by allowing disparate analysis tools to collaborate with

each other and share their information via a service platform can be highly beneficial. Not only it would enhance and speed up the work of a software engineer by giving him/her access to a big amount of information available without the need to install several tools and to cope with many output formats, but it would also promote the uncovering of new meaningful and interesting metrics and information deriving from the most diverse types of analysis that can finally “talk” to each other.

3

SOFAS Architecture

SOFAS: A Lightweight Architecture for Software Analysis as a Service
Giacomo Ghezzi and Harald C. Gall
Proc. Working IEEE/IFIP Conference on Software Architecture (WICSA), 2011
DOI: 10.1109/WICSA.2011.21

ACCESS to data stored in software repositories by systems such as version control, bug and issue tracking, or mailing lists is essential for assessing the quality of a software system. A myriad of analyses exploiting that data have been proposed throughout the years: source code analysis, code duplication analysis, co-change analysis, bug prediction, or detection of bug fixing patterns. However, easy and straight forward synergies between these analyses rarely exist. To tackle this problem we have developed *SOFAS*, a distributed and collaborative software analysis platform to enable a seamless interoperation of such analyses. In particular, software analyses are offered as RESTful web services that can be accessed and composed over the Internet. *SOFAS* services are accessible through a software analysis catalog where any project stakeholder can, depending on the needs or interests, pick specific analyses, combine them, let them run remotely and then fetch the final results. That way, software developers, testers, architects, or quality assurance experts are given access to quality analysis services. They are shielded from many peculiarities of tool installations and configu-

rations, but *SOFAS* offers them sophisticated and easy-to-use analyses. This paper describes in detail our SOFAS architecture, its considerations and implementation aspects, and the current set of implemented and offered RESTful analysis services.

3.1 Introduction

Data about software development has been primarily used for supporting activities such as retrieving previous versions of the source code or examining the status of a change request or a defect. However, studies have highlighted the value of collecting and analyzing this diverse source of data. Researchers have come up with several analyses techniques: various static and dynamic code analyses, code clone detection, co-change analysis, bug prediction, or detection of bug fixing patterns. Yet, each of these studies has built its own methodologies and tools to extract, organize and utilize such data to perform their research. As a consequence, easy and straight forward synergies between these analyses/tools rarely exist due to their stand-alone nature, their platform dependence, their different input and output formats, and the variety of systems to analyze. Therefore, despite this richness, we still lack ways to effectively and systematically share and integrate data coming from different analyses and providers.

To tackle these problems we introduced a lightweight and flexible platform called *SOFAS* (SOFtware Analysis Services) [GG08]. It offers *distributed and collaborative software analysis services to allow for lightweight interoperability of analysis tools across platform, geographical and organizational boundaries*.

Tools are categorized in our software analysis taxonomy; they have to adhere to specific meta-models and ontologies and offer a common service interface that enables their composite use over the Internet. These distributed analysis services are accessible through an incrementally augmented software analysis catalog. The main purpose of *SOFAS* is to offer a single entry point to these software analyses. A project stakeholder shall be able to pick the analyses and compose them as required to perform his investigation. Stakeholders range from software and design engineers to software test engineers, to quality assurance or project leaders.

In [GG08], we sketched the basic idea of *software analysis as a service*: getting easy access to different analyses from various tools and providers using web services. In the meantime we have experimented with a few implementations of this idea and now can present what we consider a feasible architecture for distributed analysis services. Therefore, the contribution of this very paper is the detailed presentation of the architecture for Software Analysis as a Service, its design considerations and implementation aspects, as well as the set of actually implemented and ready-to-use services based on concrete usage scenarios.

SOFAS follows the principles of a RESTful architecture [Fie00] and allows for a

simple yet effective provisioning and use of software analyses based upon the principles of Representational State Transfer around resources on the web. Our architecture is made up by three main constituents: Software Analysis Web Services (*SA-WS*), a Software Analysis Broker (*SA-B*), and Software Analysis Ontologies (*SA-Ontos*). *SA-WS* "wrap" already existing analysis tools as standard RESTful web service interfaces. The *SA-B* acts as the services manager and the interface between the services and the users. It contains a catalogue of all the registered analysis services with respect to a specific software analysis taxonomy. *SA-Ontos* define and represent the data consumed and produced by the different services.

The analyses are accessible via a single entry point and easily invokable by information such as the URLs of the source control repository, the issue tracking system, or release notes, etc. The user can then compile a "workflow" of the analyses that are required for a particular task. The *SOFAS* platform will take care of actually calling the different services and returning the final results.

Next, we will describe the constituents of the *SOFAS* architecture, its main components and their interaction. Then we will explain how we represent and structure the data produced by different analyses and domains in a homogeneous way. Finally, we will show by means of a working example how *SOFAS* actually works.

3.2 The *SOFAS* Architecture

In the past years, our group has devised many studies on software evolution and software analysis. The tools we developed and the knowledge we gained are the backbone of a software evolution analysis platform called Evolizer [GFP09]. However, while implementing it and struggling to integrate data produced by other tools for specific analyses, we realized that a big potential lies in having analyses easily accessible and composable, without platform and language limitations, and not having to install and configure particular tools.

SOFAS follows the principles of a RESTful architecture (as introduced by Fielding [Fie00]) and allows for a simple yet effective provisioning and use of analyses based upon the principles of Representational State Transfer around resources on the web. Software analyses are no longer bound to integrated development environments such as Eclipse or other IDEs, but they are accessible on the web on a common web architecture, shown in Figure 3.1. This architecture is made up by three main constituents: Software Analysis Web Services (*SA-WS*), a Software Analysis Broker (*SA-B*), and Software Analysis

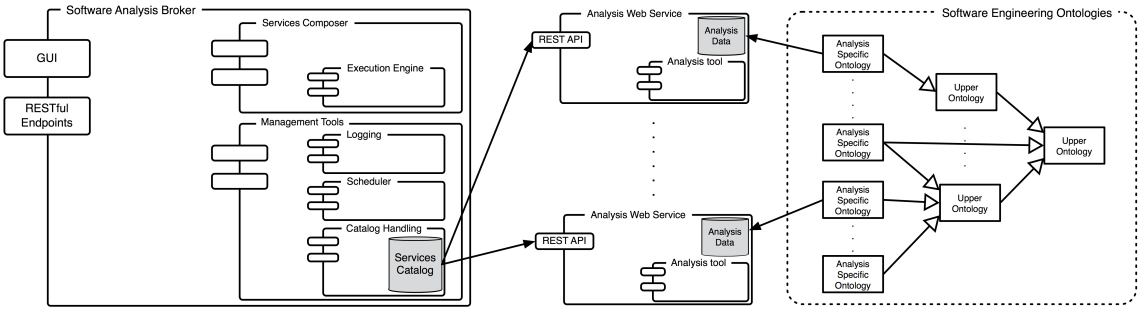

Figure 3.1: *SOFAS* overall architecture

Ontologies (*SA-Ontos*). *SA-WS* "wrap" already existing analysis tools by exposing their functionalities and data through standard RESTful web service interfaces. The *SA-B* acts as the services manager and the interface between the services and the users. It contains a catalog of all the registered analysis services with respect to a specific software analysis taxonomy. As such, the domain of analysis services is described in a semantical way, enabling users to browse and search for their analysis service of interest. *SA-Ontos* define and represent the data consumed and produced by the different services. Upper ontologies represent generic concepts common to several specific ontologies, providing semantic links between them. In the following we describe each of these three components.

3.2.1 Software Analysis Web Services

We use web services over other competing middleware technologies as it is a standard and offers many of the features we need: language, platform and location independence and ease of use. Moreover, we use a RESTful architecture since its very core properties are highly beneficial for our purposes, as we will explain.

Architectural considerations for *SOFAS*

Early prototypes of *SOFAS* were based on classic SOAP RPC-based web services. However, while they can be powerful, the rationale behind them is still highly "application dependent." They were basically created to provide web-based, language independent versions of standard applications, through the use of remote procedure calls (or remote invocations). This means that services may expose any set of operations defined with an arbitrary vocabulary of nouns and verbs, just as applications (e.g. `getUsers()`,

`getAnalysis(String analysisName)`). Moreover, since HTTP is used only as a mean of transportation, many useful HTTP capabilities, i.e. authentication, content type negotiation, caching, etc., are ignored only to then be re-invented by the service designer as specific methods, overloading the service with yet more arbitrary and heterogeneous methods. Examples are the addition of methods such as `getUsers(`**`String usersListFormat`**`), getUsersAsXML() or getAnalysis(String name,` **`String userName, String userPassword,...`**`)`).

The goal of our approach is to provide software analyses—and in particular the data produced—in a simple, generic standardized way, hiding all the peculiarities of the tools actually implementing them. The software analyses we address, typically are linear in the way they work and, more importantly, they behave almost exactly the same way: they need some information about the software project and then run the analysis (be it the code, its source code repository, etc.); once the analysis is done the data produced can be fetched in different, specific formats and the analysis data itself can be updated or deleted. The use of SOAP RPC-based web services would have thus been, in our case, a counterproductive solution, adding unnecessary complexity. The main requirements and characteristics of our services were indeed some of the main inherent principles of REST.

RESTful web services are based directly on HTTP without any additional layer or protocol. They can thus maximize the direct use of the pre-existing, well-defined interface and other built-in capabilities provided by HTTP, minimizing the addition of new application-specific features on top of it. Therefore, in contrast to classic SOAP RPC-based web services, they can use the existing, known standard rich vocabulary of HTTP methods, Internet media types, URIs and response codes. Moreover, they can also directly exploit HTTP caching, user authentication, content type negotiation, etc. To put it in a nutshell, a RESTful web service provides a uniform interface to the clients, no matter what it actually does. It is a collection of resources all identified by URIs, which can be accessed and manipulated with HTTP methods (e.g., POST, GET, PUT or DELETE). Moreover, every message exchanged is self-descriptive as it always contains the Internet media type of the content, which is enough to describe how to process it.

The example SOAP-RPC methods we showed before, in the case of RESTful services would boil down to only one HTTP method, a GET on either the URI identifying the users (e.g. GET `http://svexample.com/users`) or the analysis (e.g. GET `http://svexample.com/analysis_1`). The HTTP content type negotiation and access authentication will take care of limiting access to the right users and returning the data in the required format. The combination of specific software analysis services and REST

allows us to provide a truly uniform, standard and straight forward interface to those services.

The *SOFAS* Implementation

All services expose two types of resources: the service itself (*e.g.*, `http://seal.ifi.uzh.ch/svnImporter/analyses`) and the individual analyses (*e.g.*, `http://seal.ifi.uzh.ch/svnImporter/analyses/analysis_1`). The following methods are available on the service URI:

GET Lists all the existing analyses either in a simple XML-based list or an HTML table, depending on the requested Internet media type.

POST Creates and runs a new analysis. The new analysis URI is assigned automatically and returned by the operation.

On any specific analysis URI, the following methods are available:

GET This method behaves in two ways. If the analysis URI contains a query string, *e.g.*, `http://seal.ifi.uzh.ch/svnImporter/analyses/analysis_1?query=``<actual_query>"`, that string will be interpreted as a SPARQL [PS08] query to fetch specific data from the analysis. The result will be returned in the standard SPARQL Query Results XML Format [BB08]. This functionality is also known as *SPARQL Endpoint*. In case no query is encoded in the URI, the method just retrieves a representation of the entire addressed analysis, expressed in RDF [KC04]. We use RDF and its associated query language SPARQL, because we describe all the data produced by the analyses with ontologies. We will explain ontologies in Section 3.2.3.

HEAD Checks if the addressed analysis exists, and if so, if its data is already available. In fact, analyses can take a considerable amount of time, and thus their data might only be available upon their completion. If the analysis does not exist a NOT_FOUND (404) status code is returned, if it exists but it has not completed yet ACCEPTED (202) is returned. OK (200) is returned if it exists and it has completed successfully.

PUT Replaces the addressed analysis, or if it does not exist, creates and runs it.

DELETE Deletes the addressed analysis.

3.2.2 The existing *SOFAS* services

So far, *SOFAS* contains all the analysis services shown in the architecture overview in Figure 3.1 and a few more. They are as follows:

Version history services for CVS, SVN, and GIT They extract the version control information comprising release, revision, and commit information from CVS, SVN and GIT repositories: who changed when/which source file and how many lines have been inserted/deleted. In order to work, these services only need the URL of the repository and valid user credentials (username and password). Additional options to further fine-tune the data extraction are also available. For example, extracting the history of just a specific revision interval or fetching the content of every file revision of specific file types. The latter option is particularly useful when additional analyses need to be performed on the actual source code (*e.g.*, model extraction, metrics calculation, etc.).

Meta-model extraction service Given just the source code of a software system, it extracts its static structure in the form of a FAMIX model [TDD00] (a language independent meta-model describing the static structure of object-oriented software). The service is able to partially reconstruct the static structure even when the source code does not compile or has errors, by applying the heuristics already developed for ZBinder [PGG07].

Version history meta-model service Given a version history extracted by any version history service, it extracts the FAMIX model of all the existing or of a selected set of releases. The model reconstruction works exactly as the previous service.

Metrics service It computes the most common software metrics from a software system. This service accepts two types of inputs: raw source code or FAMIX meta-models created by the aforementioned FAMIX services. In the current version, it computes these metrics:

- Fan-In and Fan-Out of classes, methods and packages.
- McCabe's cyclomatic complexity [McC76b] of classes, methods and packages.

- Lines of code (LOC) of classes, methods and packages.
- Number of calls in the entire system.
- Height of inheritance tree of classes (HIT).
- Average hierarchy height of the entire system (AHH).
- Average number of derived classes of the entire system (ANDC).
- Number of direct sub-classes of a classes (NDC).
- Number of methods overriding a method in any one of the super-classes of a class (NORM).
- Number of classes (NOC).
- Number of packages (NOP).
- Number of attributes (static and non) of classes and packages (NOA).
- Number of methods (static and non) of classes and packages (NOM).
- Number of parameters of a method (NOPAR).

If no other piece of information is given to the service, it will compute all the metrics for all the source code entities found. Otherwise, the user can set the service to compute only specific metrics for selected entities.

Change Coupling service Given the version history of a software project, it extracts the change couplings for all the files as described by Gall *et al.* [GJK03]. This means that for every versioned file, it extracts what other files were simultaneously changed with them, how many times and when. The more two files have changed together, compared to the total number of changes they were involved, the more they are coupled.

Change type distilling service Given a project version history (extracted by one of the aforementioned services), it extracts, for each revision, all the fine-grain source code changes of each source code file. These changes are then classified following the change types taxonomy proposed in [FWPG07]. The algorithms used to extract these changes are also based on the ones developed by Fluri *et al.* in the aforementioned paper for the original Change Distiller tool [GFP09].

Issue tracking history services for Bugzilla, Google Code, Trac, and SourceForge They extract all the historical issue tracking information (problem reports and change requests) from a given issue tracking repository. This data is usually used as-is or together with the project version control information. In the first case, it can help assess the average bug-fixing time, the distribution of bug severity, etc. In the second case, it can be used for more complex analyses, such as location of fault prone files, location and analysis of bug fixing changes, bug prediction, etc. As for the version control services, also this one can be set to import just a range of issues, instead of the whole history.

Issue-revision linker services Given the issue tracking and version histories of a specific software project, they reconstruct the links between issues and the revisions (also known as commits) that fixed them. As of now three of these services exist for the different heuristics proposed by Mockus *et al.* [MV00], Sliwerski *et al.* [SZZ05b], and Fischer *et al.* [FPG03b].

Note that all these services structure the extracted data following specific ontologies, which we explain next.

3.2.3 Software Analysis Ontologies

We described how REST provides us a truly uniform interface to describe all the analysis services in our architecture, the structure of their input and output and how to invoke them at a syntactic level. However, there is no way to programmatically know what a service actually offers and what the data it consumes/produces means. We address this problem by exploiting semantic web technologies, in particular OWL. An ontology is a formal description of the important concepts (classes of objects) identified in the domain of discourse and their relationship to one another [Gru93]. It provides a common vocabulary for a specific domain, which can be used to express the meta-data needed to capture the knowledge of the exchanged, shared or reused data. Ontologies help tackling both problems, *i.e.*, meaningful service descriptions and data representation.

With OWL we can assign input and output data a clear semantics and a precise syntax, as it is a standardized XML based language. It offers some highly beneficial ontological properties: (1) heterogenous domain ontologies can be semantically "linked" to each other by means of one or more upper ontology, describing general concepts across a wide range of domains. In this way it is possible to reach interoperability between a large number of ontologies accessible "under" some upper ontology. In terms of software analysis

services, it means that results from disparate types can be automatically combined given that they share some common concepts; (2) the OWL Description Logic foundation enables automatic reasoning and derive additional knowledge; (3) we can use a powerful query language such as SPARQL; and (4) in contrast to XML and XQuery, which operate on the structure of the data, OWL treats data based on its semantics. This allows for an extension of the data model with no backwards compatibility problems with existing tools.

To describe the data produced by software analyses we developed our own family of Software Evolution ONtologies (*SEON*). The goal is to describe in a clear and univoque way different aspects of software and its evolution, such as version control, issue tracking, static source code structure, change coupling, software design metrics, etc. Figure 3.2

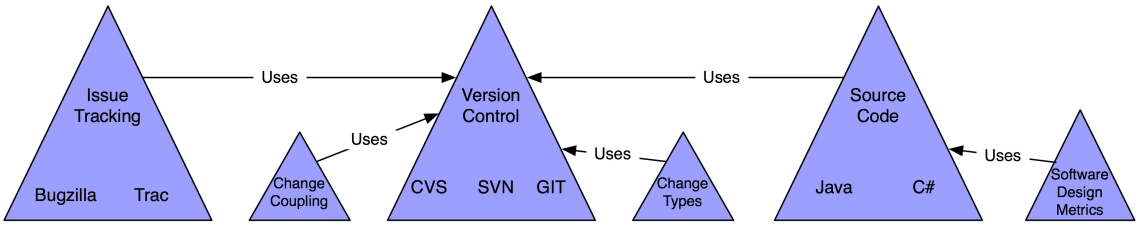

Figure 3.2: *SEON* overall structure.

depicts the basic structure of *SEON*. As of now, the domains described are only the ones addressed by the existing analysis services. For each of the three major subdomains (represented as individual ontology pyramids) we have developed higher level ontologies defining their common concepts. For system-specific or language-dependent concepts we developed some concrete low-level ontologies. The different ontologies share some concepts and properties. More specifically, the source code, issue tracking, change types and change coupling ontologies use concepts of the version control system one, as the metrics ontology does from the source code one. The version control pyramid can be thus considered the core of *SEON* as it interconnects the three major subdomains. The *issue tracking ontologies* (for CVS, SVN and GIT) add only few additional concepts to the generic ontology. The SVN ontology, for example, adds the concepts of copies, moves and renames as these operations are poorly supported (or not at all) by others systems. The system specific ontologies introduce additional concepts as the two systems have a different way of classifying bug and issue priorities. Moreover some systems might have a slightly richer or different issue description. For example, Bugzilla keeps track of OS and hardware under which the issue was experienced while Trac does not.

The *source code ontology* models all the static source code structures based on the FAMIX meta model. We decided to use FAMIX instead of other meta models such as UML, as it has a finer granularity and offers more details. As FAMIX was already devised as a language independent source code model for OO programming languages, we represent all the important concepts in the generic ontology. We created the Java and C# ontologies just to address the few particularities. The central concept of this ontology is the class. Through a class the source code ontology can be linked to a version control history, as classes are contained in versioned files.

The *metrics, change types and change coupling ontologies* are simple as they describe rather basic and unstructured data. The first one classifies common software product metrics such as [CK94, McC76b]. All these metrics are either computed at package, class or method level. The concept of a metric itself is associated to the concept of an entity of the source code ontology, which represents both classes and methods. The change types ontology describes the source code change types according to the taxonomy proposed by Fluri *et al.* [FWPG07]. The change coupling ontology describes how intense two files are coupled in a project: how many times they were changed together during a specific release period.

SEON is continuously evolving. We envision many other ontologies to be incrementally added as new services are provided, for example, ontologies targeting other code quality measurements such as code clones or code smells. Furthermore, ontologies might describe different version control systems (*e.g.*, Mercurial), issue tracking systems (*e.g.*, Jira, Mantis) and programming languages (*e.g.*, C++, Eiffel, Python). The expansion and referencing of existing ontologies or the creation of new ones can be done without changing the already existing ontologies. This is due to the nature of semantic web ontologies: a continuously growing distributed network of loosely interlinked, expandable ontologies. Figure 3.3 sketches three of the ontologies we just introduced.

3.2.4 Software Analysis Broker

Web services enable the sharing, using, and combining the different analyses through the net. But they need to be kept track of, classified in a registry, queried, monitored and coordinated. The Software Analysis Broker (*SA-B*) takes care of that, so that the user does not have to interact directly with the raw services. As shown in Figure 3.1, the *SA-B* is made up of four main components: the *Services Catalog*, a series of management tools, the *Services Composer*, and a user interface.

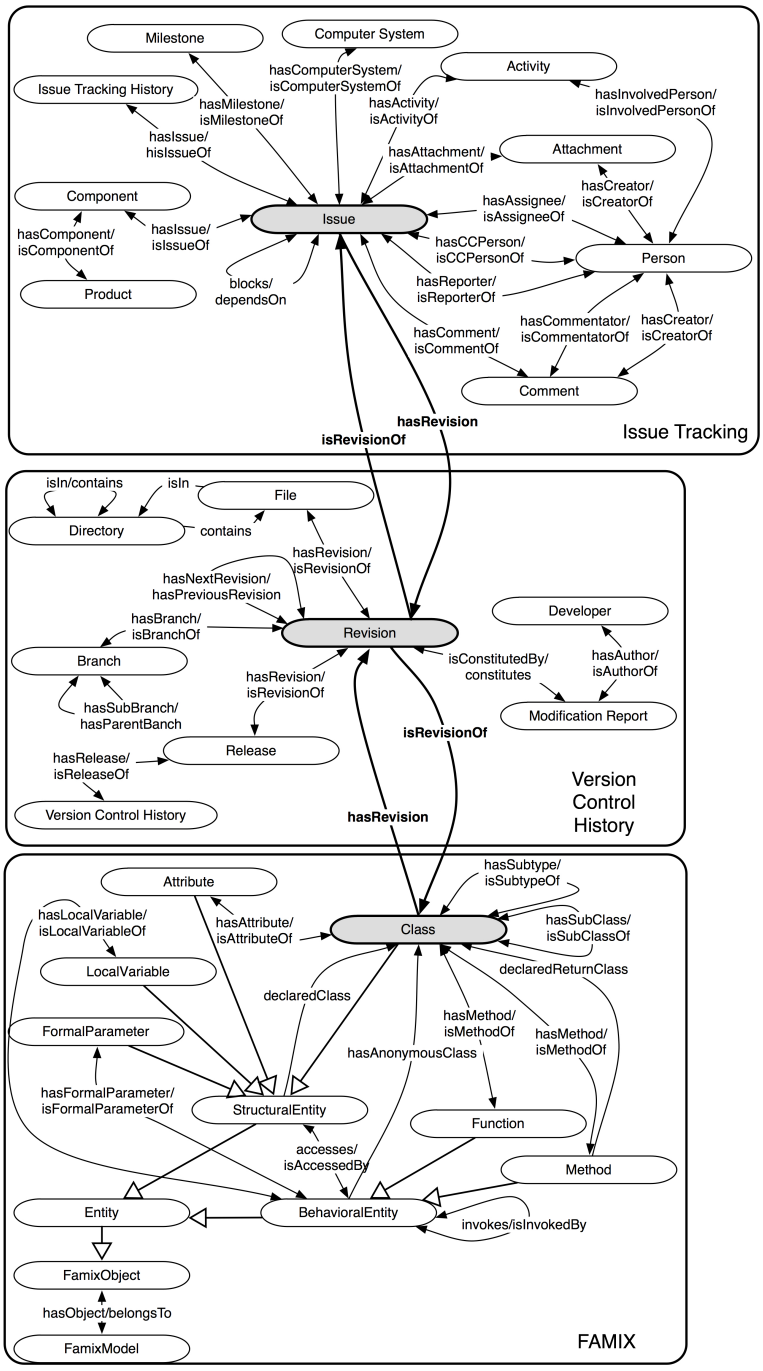

Figure 3.3: Overview of three of the major *SEON* ontologies.

User Interface

The UI is the actual access point to the *SA-B*. It consists of a web GUI, meant for human users and a series of RESTful service endpoints to be (semi)-automatically used by

applications. Through the UI the user can easily browse through the *Services Catalog* to check for analyses offered and to select some of them. Apart from the catalog, the user can also pick from some already predefined combinations of analysis services provided as high level analyses workflows (called *analysis blueprints*). Once the desired services are selected, the user might need to set some service-specific settings. Moreover, if the user chose to combine two or more services into a workflow, she would need to actually define how to do that. That is, what their sequence is, what output of a service should be fed as input to another service, etc. The user interface offers an intuitive, high level way to do that, allowing the user to combine the services in a "pipe and filter" fashion. The real composition of those services into an executable workflow and its execution is then taken care of by the *Services Composer*.

Services Catalog

The *Services Catalog* stores and classifies all the registered analysis services so that a user can automatically discover services, invoke them, and fetch the results. To do that, an unambiguous classification is essential. We developed such a specific software analysis taxonomy to systematically classify existing and future services. This taxonomy divides the possible analyses into three main categories: development process, underlying models, and source code.

Software development analyses are subdivided into those targeting the development history (extraction, prediction and analysis of source code changes and bugs), its underlying process, and the teams involved in it (their dynamics and metrics). *Model analyses* include those targeting the extraction, either dynamic or static, of specific behavioral and structural model representations (UML, FAMIX, call graphs, etc.) and those computing differences between two models. *Code analyses* are further divided into categories such as checking code well-formedness, correctness and quality. For example, the code quality category is then split into subcategories dealing with code security, conciseness, performance, and design. The latter contains, among others, extractors and analyzers of design metrics and code-smells. A full description is beyond the scope of this paper, but for more details we refer to the *SOFAS* website[1].

Since the literature lacks a preexisting taxonomy of this kind, we structured it mainly using the currently existing approaches as a blueprint and so that they would "fit" reasonably well. This means that our *Services Catalog* is one possibility and by no means

[1] https://seal.ifi.uzh.ch/sofas

complete, as in any classification there are always individuals that do not clearly fit in any category or fit in more than one. However, the proposed categories are reasonable enough, in particular from the perspective of a user who wants to find some particular analyses without struggling with many and sometimes obscure categorizations. Our taxonomy is defined as an OWL ontology. Thus the catalog itself ends up being an instance of that ontology and every registered service an instance of a specific class of that ontology. The ontology is managed and stored in a triple-store and accessed using JENA[2], an open source framework meant exactly to allow for the querying, storing and analysis of RDF/OWL data through a high-level, intuitive API.

We decided to develop this lightweight semantic web-based custom solution, instead of using UDDI, the standard solution for web service registries, for several reasons. The most prominent are related on how it deals with taxonomies, how they are defined, how they are used to classify and then fetch services. UDDI's taxonomies are usually rather simple, flat and with a convoluted definition, especially compared to the cleanness and richness one can reach by using OWL. This highly affects the quality and broadness of classifying and subsequently querying services. On the other hand, with OWL the classification can be as complex and specific as we want the taxonomy to be. Powerful query languages such as SPARQL can be used to query the catalog and fetch specific services. With these languages, the querying options become manifold: services can then be queried based on what categories they belong to, on any of their attributes, on the attributes of any of the categories they belong to, etc.

Services management tools

Typically just calling services or combining them is not enough. In particular, this holds for long running, asynchronous web services. They need, for example, to be logged and monitored to check if they are up and running, if they are in an erroneous state and why, if they have completed a required operation, etc. Even though these functionalities are vital for end users, their use should be as transparent, standardized and automated as possible. Thus, we implemented a series of services that take care of implementing that as services. As a result, calls to them can be easily weaved into a user defined workflow. The *Services Composer* takes care of doing that.

[2]http://jena.sourceforge.net/

Services composer

This component works both as an interpreter and as the engine running the services workflows. It translates the high level service composition workflow defined by a user through the UI into executable processes and runs them. The decoupling between the user composition definition and the actual composition language is useful for two reasons. First, it allows the user to compose services in a intuitive way, hiding the complexity and technicalities of the actual composition and orchestration. Second, calls to additional services can be automatically weaved into a user defined workflow. In our case, the *Services Composer* adds calls to the management services we just introduced.

We decided to rely on our own custom service composition language and execution engine instead of using existing standards—such as WS-BPEL [JE07a] and one of its related engines—for several reasons. The most prominent is that these languages are meant to be used for SOAP RPC-based services, defined using WSDL. A standard description language (called Web Application Description Language, WADL [Had09]) has only been recently proposed. Most RESTful services still rely on just human-oriented documentation. Thus, it comes as no surprise that no standard composition language exists yet. Custom solutions such as extending BPEL to account for REST [Pau08], describing RESTful services with WSDL 2.0 or creating new ad-hoc languages and tools [Pau09] have been recently proposed. However, they have not really gained ground or have been used outside theoretical case studies.

As shown in Section 3.2.1, the services in our architecture not only have the same interface, but they also exhibit the same behavior. Analyses can be started, managed and the outcome data be fetched always in the same manner. This allows us to make several assumptions and simplifications in modeling how analyses work and how they can be composed. A full-blown approach based on BPEL or on a BPEL-like solution would thus be counter productive, adding unnecessary complexity. In particular, an analysis services workflow always consists of starting one or more analyses (an HTTP post method on the service URL), waiting for them to finish (Repeatedly calling an HTTP head method on the analysis URL) and, when done, passing the URI of the results to waiting analyses (along with analysis specific options) and so on, until the workflow is completed, as shown in Figure 3.4. Consequently, our composer only needs to be able to fetch the services the user selects from the catalog, create the actual service calls, interweave between the simple control structures (*i.e.*, loops to wait for an analysis to complete, loops to restart erroneous analyses or error handling procedures) and pass the data produced by a service to any

waiting service in a classic “pipe and filter” fashion.

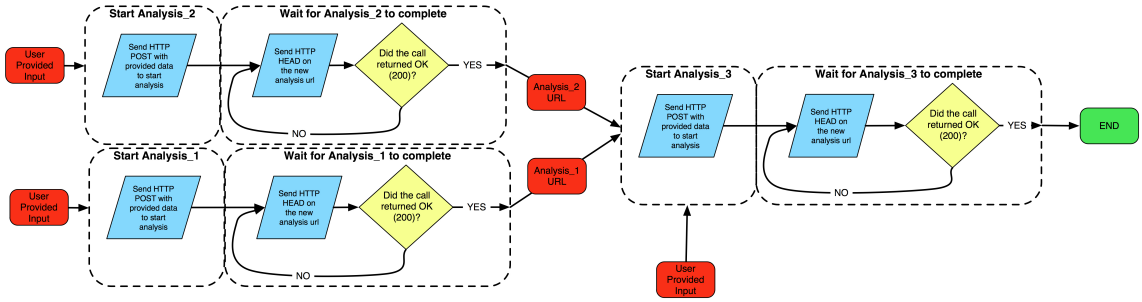

Figure 3.4: An example of a software analysis workflow.

Our solution is based on the use of WADL to describe the analysis services. By reading a service WADL, the composer knows the input data needed and can thus ask the user to provide it. WADL allows also to incorporate textual descriptions of a service, of its methods and their parameters. This is especially useful for human users. Moreover, we slightly expanded the WADL description so that input and output, when needed, can be declared as being described by specific ontologies, in our case the *SA-Ontos* we previously introduced. This is inspired by SAWSDL (Semantic Annotations for WSDL) [FL07]. Not only this is useful to guide the user in the composition, but the composer itself can, once an analysis is picked, suggests additional services to add to the workflow. In fact it can browse the catalog to fetch all the analyses that produce or require that data. For example, if the service chosen required version control history data as input, all the version control history services would be suggested.

The workflows, once created, will then be stored so that they can be rerun and/or modified in the future. These workflow themselves are RESTful services, adhering to the common behavior we outlined earlier in Section 3.2.1. That means that they can then be interacted with as any other analysis service in *SOFAS*. The only difference is that they require as input all the data needed to invoke correctly all the services in the workflow and producing as output the data generate by the services closing the workflow. Also a WADL description will be created for them. In this way, they can also be composed with other services, into even more complex and structured workflows.

In the following, we show a first validation consisting of a usage scenario and actual systems that have been analyzed with *SOFAS* services.

3.3 Validation

Let us show how *SOFAS* can support a user in a concrete software quality analysis task: finding the code smells of the major releases of ArgoUML[3]. Code smells allow one to spot abnormal and suspicious code entities but also to get an overall impression of the system, as shown, for example, by Lanza and Marinescu [LM05]. Moreover, tracking them over a project history helps in assessing the overall quality evolution.

The data to start any analysis involving a system's source code and its history lies in its version control repository (SVN in our case). The first step is thus the invocation of the *SVN version control history service* to extract the full history of the project (since early 1998) along with the source code for all the releases it finds. Once completed, the link to the analysis then is passed to the *Version history meta-model service*. Based solely on that, the service, knowing it is a link to a version control history, is able to automatically fetch the list of releases found and, for each of them, get their source code and reconstruct their FAMIX meta-model (their static structure). The links to each of these models are then passed to the *Metrics service* that, based on each of them, computes the metrics we introduced in Section 3.2.2. These metrics can then be combined to detect smells such as God Class, Feature Envy, etc. As of now, the user has to manually do this last step. However, it is rather a straight forward task, as it is just a matter of combining some of the provided metrics. Nevertheless, an additional service that does that automatically is currently being developed. This service will return, for each code smell the list of code entities (classes and methods) affected. Note that data produced can then also be re-used and fed into other additional services. In our example, the extracted version control history could then be passed to the *Change Coupling service* to find out which classes and files are evolutionary coupled and thus point to other possible architectural weaknesses [DLL09, GJK03].

SOFAS has already been used internally in our research group for several studies. One Microsoft Surface application uses the data produced by the *meta-model service* for purposes of multi-touch enabled code navigation and design recovery. Another application uses exactly the workflow introduced to visualize multiple evolution metrics as proposed by Pinzger *et al.* [PGFL05] on a multitouch screen. For these tools and their evaluation, some of the most popular Java-based open source projects have been analyzed (*e.g.*, ArgoUML, Eclipse, Vuze, jUnit, Tomcat, Derby). Some of the services in *SOFAS* have also been used extensively by external research groups. In particular, all the *version control*

[3] http://argouml.tigris.org/

history services were used to extract the histories of around 100 open source projects. These projects were a mix of the most known and successful ones (*i.e.*, Python, Gimp, Ruby, or OpenOffice) and the most popular projects in the major OSS forges (*i.e.*, Github, Sourceforge and Tigris). This huge amount of data has been used, for example, to study factors of success and failure in open source projects. More recently it has been used as a base to study contribution and collaboration patterns in OSS projects.

Due to space limitations and to the fact that some of those studies are still yet to be published, the list of all the projects analyzed and details of these studies cannot be fully disclosed at this point.

This is by no means a complete validation of *SOFAS*. However, we claim that it is a serious first proof of the usefulness of the proposed architecture. Furthermore, its already varied and heterogenous usage is a testimony to its versatility, not only for software engineering related tasks. As a matter of fact some of the current users come from very different backgrounds, such as Physics, Management and Economics. More in-depth validations with complex workflows and different usage scenarios will follow, as well as experiments on the properties of services in terms of run-time aspects, data volumes, and reliability.

3.4 Related Work

There is a plethora of research works exploiting software project data for software evolution. Approaches focusing on the software evolution either study its source code change history [ZWDZ04, NB05], bug history [KZJZ07], its underlying dynamics [AHM06, MH02] or a combination of them [BWKG05, GFP09]. However, all these approaches rely on their own ad-hoc developed tools and techniques and none targeted the issue of using and composing different, independent analyses. Moreover, none of them address the issue of facilitating the analysis usage by thirds by means of web services or similar technologies.

Jin and Cordy [JC05] were so far the only researchers to study a solution to these issues. They propose an ontology based software analysis tool integration system that employs a domain ontology and specifically constructed external tool adapters. They use a service-sharing methodology that employs a common domain ontology defining the conceptual space shared by the different tools and specially constructed external tool adapters, that wrap the tools into services. They also implemented a proof of concept with three reverse engineering tools that allowed them to explore service-sharing as a

viable means for facilitating interoperability among tools. We share with them the overall concept, but at the same time, the two approaches have many differences due to their partially distinct goals. In fact, the objective of their integration effort was to be able to apply a functionality/analysis available in one tool to the fact-base of another one in a very simple way. For this reason, they used a domain ontology just to describe the set of representational concepts that the different tools to be integrated require and support. On the other hand, our goal is to offer a much broader and versatile solution. In fact, we intend to exploit ontologies on a much broader scale: to catalog and describe the services, to represent and standardize their input and output accordingly to the type of analysis offered, to semantically link different results and to perform (semi)-automatic reasoning on them. Moreover their paper just sketches the overall rationale of the approach without going into details on how the proposed architecture was actually implemented and which technologies were used.

The use of web services and semantic web technologies for software analysis, and software engineering in general, has only just recently been addressed in research by just a few works. These works all have focused on providing ontologies to representing software analysis data and concepts to foster software reuse and maintenance. For example, generic software engineering concepts (classes, tests, metrics, requirements, etc.) [HWCK06], higher level meta-data about software components (e.g. the programming language, licensing models, ownership and authorship data) [HKST06]. More related to our approach, Kiefer *et al.* [KBT07], developed a software repository data ontology including software, release and bug related information based on based on Evolizer's [GFP09] data models. However none of these models, are then used for concrete software engineering tasks other than a small proof of concept.

3.5 Conclusions and Future Work

In this paper we presented *SOFAS*, a flexible and lightweight architecture—both in terms of resources and knowledge requirements—to enable the use and combination of software analyses across platform, geographical and organizational boundaries. We devised these analyses as RESTful webservices accessible through a software analysis broker where users can register, share and use their tools. To enable (semi)-automatic use and composition, these services are classified and mapped into a software analysis taxonomy and adhere to specific meta-models and ontologies for their category of analysis.

We claim that an architecture like the one we devised is highly beneficial for the field of software (evolution) analysis. With very few actions, simple, common analyses can be combined into new, complex and structured ones and then ran. In this way, different stakeholders—which we introduced in the introduction to this paper—could easily extract different type of interesting and useful data about a software project. A project leader might be able to check the status and health of the project by checking, for example, the amount of bugs per file, their distribution and their lifetime (how long it takes to fix them) to maybe allocate resources where needed. A software quality assurance engineer might use it to check the quality of the code (*e.g.*, with OO metric and clone detectors) and be sure that no "software rotting" is going on, or fix it before it goes out of control. A software engineer might use it on a re-engineering task to extract change coupling between source code files (and their evolution) to detect possible cross cutting concerns, hidden or forgotten business rules, clones or in general classes to be improve code cohesion. He might also use it to extract source code metrics to detect potentially problematic classes or disharmonies (*e.g.*, God Class, Brain Class, Intensive Coupling) to drive future refactoring.

SOFAS is still a work in progress and in continuous evolution. In particular, the service composition is still in an early phase. Service composition into workflows is for now only possible through the *SOFAS* web UI provided, and thus only available for human users. In the future we plan to develop a simple ad-hoc composition language so that workflows can be programmatically sent for execution to *SOFAS*. Moreover, new services will be constantly added as soon as they will be developed. Our research group is the main driving force behind it and the provider of the entire support infrastructure and services. However we recently started to look actively for collaborations with other groups to sharing their knowledge and their tools and thus provide new services to the architecture. This is vital in asserting the success and usefulness of our architecture, as one of its main foundations is indeed the sharing of new and diverse analyses by means of services.

4

SEON: A Pyramid of Ontologies for Software Evolution

SEON: A Pyramid of Ontologies for Software Evolution and its Applications
Michael W´ursch, Giacomo Ghezzi, Matthias Hert, Gerald Reif and Harald C. Gall
Computing Journal (accepted for publication, 2012)

DOI: 10.1007/s00607-012-0204-1

THE Semantic Web provides a standardized, well-established framework to define and work with ontologies. It is especially apt for machine processing. However, researchers in the field of software evolution have not really taken advantage of that so far. In this paper, we address the potential of representing software evolution knowledge with ontologies and Semantic Web technology, such as Linked Data and automated reasoning. We present *SEON*, a pyramid of ontologies for software evolution, which describes stakeholders, their activities, artifacts they create, and the relations among all of them. We show the use of evolution-specific ontologies for establishing a shared taxonomy of software analysis services, for defining extensible meta-models, for explicitly describing relationships among artifacts, and for linking data such as code structures, issues (change requests), bugs, and basically any changes made to a system over time. For validation, we discuss three different approaches, which are backed

by *SEON* and enable semantically enriched software evolution analysis. These techniques have been fully implemented as tools and cover software analysis with web services, a natural language query interface for developers, and large-scale software visualization.

4.1 Introduction

Scientia potentia est. Knowledge is power. For millennia this maxim has been valid, and will likely remain so in the future—even in an age of information overload, where the entire humankind produces roughly two zettabytes data a year.[1]

This also holds for the domain of software engineering, where even small development teams accumulate gigabytes of interdependent artifacts over the years. They are stored in software repositories, such as version control systems, issue trackers, but also in Wikis, and even mailing lists. Understanding what factors distinguish successful development projects from others is key to improve the quality of software systems. Distilling the knowledge of best practices from random noise found in a software repository is what the field of software evolution research and mining software repositories aims for.

But data is not necessarily information, and information not necessarily knowledge. Successful differentiation requires understanding of data semantics and interpretation. The obvious solution to this dichotomy is that machines and humans form a joint-venture: humans define the semantics and machines bring in their computational power for the advent of the next generation of software evolution support tools. The Semantic Web provides the instruments to achieve such a synergy; ontologies created by human beings represent knowledge and give semantic meaning to raw data so that machines can automatically process and exchange it. Reasoners make implicit knowledge explicit by inferring relations that were previously missing. Interestingly, these technologies yet struggle to find a wide adoption in the field of software evolution research, whereas, for example in life sciences, many applications have demonstrated the value of the Semantic Web for processing and sharing large corpora of information (*e.g.*, in [KSG+10]).

In this paper, we pursue the research question, how we can adequately describe software evolution knowledge by means of ontologies. This includes knowledge about stakeholders, activities, artifacts, and the relations among all of them. The ultimate goal is to provide software engineers with effective tool-support for managing software systems over their entire life-cycle.

The contributions of our paper are threefold:

1. We critically reflect on the potential that the Semantic Web yields for software evolution. In particular, we show four characteristics that are most beneficial for

[1] According to the study "Digital Universe: Extracting Value from Chaos" by IDC, humans created 1.8 zettabytes data in 2011. This value is estimated to double every two years.

the field: shared taxonomies, extensible meta-models, explicit relations, and Linked Data.

2. We present *SEON*, our family of software evolution ontologies. These ontologies describe knowledge on multiple levels of abstraction ranging from code structures up to stakeholder activities.

3. We describe three semantics-aware tools that make extensive use of *SEON* and help developers in dealing with large amounts of software evolution data: software analysis with web services, a natural language query interface for developers, and large-scale software visualization. All three of them have been fully implemented for a proof-of-concept.

In the remainder of this paper, we will describe the potential of Semantic Web technology for dealing with software evolution.

In Section 4.2, we give a brief overview on the Semantic Web and related technologies, before we discuss in Section 4.3 the advances they can bring to the field of software evolution research. We also address a set of general challenges yet to be solved before the full potential of Semantic Web-enabled approaches can be realized.

At the core of this paper is *SEON*, our pyramid of ontologies for software evolution, which is described in Section 4.4. These ontologies provide a taxonomy to share software evolution data of various abstraction levels across the boundaries of different tools and organizations.

In Section 4.5, we describe three different applications of *SEON* from three distinct domains to showcase the utility and versatility of ontologies in the context of software evolution research. A selection of other ontology-driven approaches in the field of software engineering is discussed in Section 4.6. In Section 4.7, we conclude the paper.

4.2 The Semantic Web in a Nutshell

Berners-Lee *et al.* define the Semantic Web as *"an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."* [BLHL01]

Despite its origins, the Semantic Web is not limited to annotating webpages with meta-data. Virtually any piece of knowledge can be described in a computer-processable way by defining an ontology for the domain of discourse. An ontology formally describes

the concepts (classes) found in a particular domain, as well as the relationships between these concepts, and the attributes used to describe them [Gru93]. For example, in the domain of software evolution, we define concepts, such as *User*, *Developer*, *Bug*, or *Java Class*; relationships, such as *reports bug*, *resolves bug*, or *affects Java Class*; and attributes, such as *email address of developer*, *resolution date of bug*, *severity of bug*, etc.

Since the Semantic Web describes knowledge based on formal semantics, data can be exchanged among two applications that support the same ontology, even if they were not meant to interoperate in the first place. The data representation format no longer needs to be custom-tailored to a specific task, but can be re-used later.

Researchers and practitioners came up with a number of standards, W3C recommendations, development frameworks, APIs, and databases to pursue the vision of the Semantic Web. The Resource Description Framework (RDF) [KC04] is the data-model for representing meta-data in the Semantic Web. The RDF data-model formalizes meta-data based on *subject – predicate – object* triples, so called RDF statements. RDF triples are used to make a statement about a resource of the real world. A resource can be almost anything: a project, a bug report, a person, a Web page, etc. Every resource in RDF is identified by a Uniform Resource Identifier (URI) [BLFM98].

In an RDF statement the subject is the thing (the resource) we want to make a statement about. The predicate defines the kind of information we want to express about the subject. The object defines the value of the predicate. In the RDF data-model, information is represented as a graph with the statements as nodes (subject, object) connected by labeled, directed arcs (predicate). The query language SPARQL [PS08] can be used to query such RDF graphs.

RDF itself is domain-independent in that no assumptions about a particular domain of discourse are made. It is up to the users to define specific ontologies in an ontology definition language, such as the Web Ontology Language (OWL) [De04]. OWL enables the use of description logic (DL) expressions to further describe the relationships between classes and to restrict the use of properties [PSHe04]. For example, two classes can be declared to be disjoint, new classes can be built as the union/intersection of others, or the cardinality of a property can be restricted to define how often a property can be applied to an instance of a class. OWL can describe both uniformly, data schema and instance data.

In addition to the W3C recommendations, the Semantic Web community developed tools to process RDF meta-data. Jena[2] emerged from the *HP Labs Semantic Web Program* and recently became an Apache incubator project. It is a Java framework for building

[2]http://incubator.apache.org/jena/

applications for the Semantic Web and provides a programmatic environment for RDF and OWL. Reasoners, *e.g.*, Pellet[3] or HermiT,[4] infer logical consequences from a set of asserted facts or axioms. RDF databases, such as Sesame[5] or Virtuoso,[6] store RDF triples and can be queried with SPARQL.

4.3 The Potential of Ontologies in Software Evolution Research

Over the last decade, software evolution research brought up various tools that help engineers to better deal with large, ever-changing legacy systems. In [MWG10] it was argued that most of these tools use proprietary data formats to store their artifacts, which hampers tool-interoperability. Furthermore, querying software evolution knowledge is difficult, especially when queries span across different domains. Queries such as *"In which release was this bug fixed and which source code modifications where done to fix it?"* involve several domains (*i.e.*, static source code, version control, issue tracking), something which is not originally supported by common software repositories.

The *Mining Software Repositories*[7] community tackled this issue by mirroring software artifacts from various sources in a central (relational) database [DGLP08]. This gave rise to numerous experiments where researchers successfully mined such databases for interesting patterns (see [KCM07] for an overview; specific examples can be found in [BWKG05, FPG03a, GFP09, SZZ05b]). Unfortunately, such a central database imposes a universal data schema onto all contributing tools, turning the software repository into a rigid and inflexible monolith.

Semantic Web technology has been designed as a solution to such integration problems. In the following, we briefly revisit the characteristics of the Semantic Web that we identified in in our previous work to be most beneficial for the field of software evolution research.

[3] http://clarkparsia.com/pellet/
[4] http://hermit-reasoner.com/
[5] http://www.openrdf.org/
[6] http://virtuoso.openlinksw.com/
[7] http://www.msrconf.org

4.3.1 Establishing a Shared Taxonomy of Software Evolution

One of the critical design aspects when building a knowledge base is to define a meta-model that describes the knowledge in an adequate level of detail. To share data among different tools, they need to understand the same vocabulary.

In practice, there are a number of general-purpose meta-models in software engineering, such as the Dagstuhl Middle Metamodel (DMM) [LTP04], as well as more specific ones, *e.g.*, for source code. Many of them define the same concepts, but name them differently. The *C++ Data Model* [CGK98] of Chen and the *FAMOOS Information Exchange Model (FAMIX)* of Tichelaar *et al.* [TDD00] can both be used to describe source code written in C++. Although they share many commonalities, tools written to work on FAMIX cannot process instances of Chen's model and vice versa, *e.g.*, to replicate experiments. Further, meta-models are often implemented in terms of a relational database schema. Exchanging schemata among different databases, however, is relatively inconvenient, due to vendor-specific implementations of data definition languages. Instead, and despite the advent of specialized exchange formats, such as RSF [MK88], XMI [Obj98], or GXL [AWR02], data is often serialized into plain XML or a comma separated value (csv) format. These formats are not semantics-preserving and therefore of limited use.

While relational database schemata are hardly ever exchanged, ontologies were explicitly designed to be shared. They can be serialized using the RDF/XML standard and exchanged without loss of data semantics. In Section 4.4, we propose our set of ontologies that provide a taxonomy for important concepts in the domain of software evolution. With the approach described in Section 4.5.1, we demonstrate how such taxonomy fosters interoperability between an entire ecosystem of software services.

4.3.2 Defining Extensible Meta-Models

Especially in a research context, meta-models tend to evolve constantly. Therefore, they need to be designed to be extensible. For example, adding data about additional software artifacts should be straight-forward and possible without breaking applications that rely on the original model.

When meta-models are extended, this usually enforces database schema changes—a time consuming operation, as the whole repository and all database keys have to be reorganized. Chances are more than likely that existing applications directly accessing the

database will break in such a case.

Designing ontologies is comparable to designing Entity-Relationship or UML models. The result is a data schema. In the Semantic Web, however, the schema itself is described in terms of RDF triples, making it more flexible to changes than the relational one. No distinction between data and ontology is necessary, as both are simply additions or deletions of triples. It is therefore unproblematic to add more ontologies and to specialize existing concepts and properties by deriving sub-concepts and sub-properties.

In Section 4.5.2 we present a query approach that especially benefits from the extensibility of ontologies, as well as from the fact that data and meta-data are represented uniformly. Our query system analyses both, the data and meta-data and uses the results to guide developers in composing and executing queries related to program comprehension tasks. When we add new ontologies to *SEON*, our query system is able to deal with this additional knowledge without requiring us to change a single line of code.

4.3.3 Making Relations Explicit

There is no consistent way to get the meaning of a relation in relational databases. In fact, a query can join tables by any columns, which match by datatype—without any check on the semantics. While humans can often guess the meaning of a relation, computers can not. They need to be supplied with additional information. It is therefore necessary to encode a significant amount of implicit knowledge into applications to make use of the data. To search in an existing repository, or to build an own tool on top of it, researchers need to be aware of, and understand this implicit semantics.

The SPARQL query language allows one to query explicitly for relations among resources. Such queries are impossible in the relational and in the object-oriented paradigm unless relationships are explicitly mapped to tables or, in the case of object-orientation, modeled as association classes. The latter, however, can make them difficult to distinguish from “real” classes. Given the high importance of relationships in software evolution, it is preferable to model them as first class objects—which is exactly what the Semantic Web does.

The importance of this aspect is emphasized in Section 4.5.3. There we introduce our recommender tool, which depends on the explicit semantics of ontologies. Given a set of data, it searches for certain types of individuals, as well as for their relations, to recommend appropriate visualizations.

4.3.4 Linked Software Evolution Data

With only relational database technology, synergies between research tools are hard to exploit. For example, we cannot simply establish connections between data stored in two different software repositories, such as a version control system and an issue tracker. The reason for this is that it is impossible to set a link from one repository to another—relations are local, not universal. Cross-domain queries spanning multiple repositories are impossible.

One of the driving forces behind the Semantic Web is the basic assumption that data becomes more useful the more it is interlinked with other data. The simple but powerful concept of statements represented by triples of URIs can be used to build an internet-scale graph of information because it makes it possible to link and query data that is stored in different locations.

The software analysis services described in Section 4.5.1 manage data based on these principles. URIs are assigned to every artifact analyzed and all the results generated. These URIs are de-referenceable over the Web and allow services to request from other (remote) services information about resources on an as-needed basis. Like that, the software analysis services already operate on a global graph of software evolution data today.

In the next section, we describe *SEON*, an ontological description of the domain of software evolution. It exploits the characteristics of the Semantic Web mentioned above to support a wide range of semantics-aware applications.

4.4 SEON – A Pyramid of Ontologies for Software Evolution

The acronym *SEON* stands for *Software Evolution ONtologies* and represents our attempt to formally describe knowledge from the domain of software evolution analysis & mining software repositories. However, in contrast to many other existing ontologies, we did not aim to capture as much of the domain under discourse as possible. Instead, we originally incorporated only a limited set of concrete concepts and extended the ontologies solely when it was actually required by a particular analysis or by a tool that we had already built or used. Three of these tools are detailed in Section 4.5. We then followed a bottom-up approach and, from these very concrete concepts, iteratively added abstractions and extended our ontologies. This process is briefly described in Section 4.4.6.

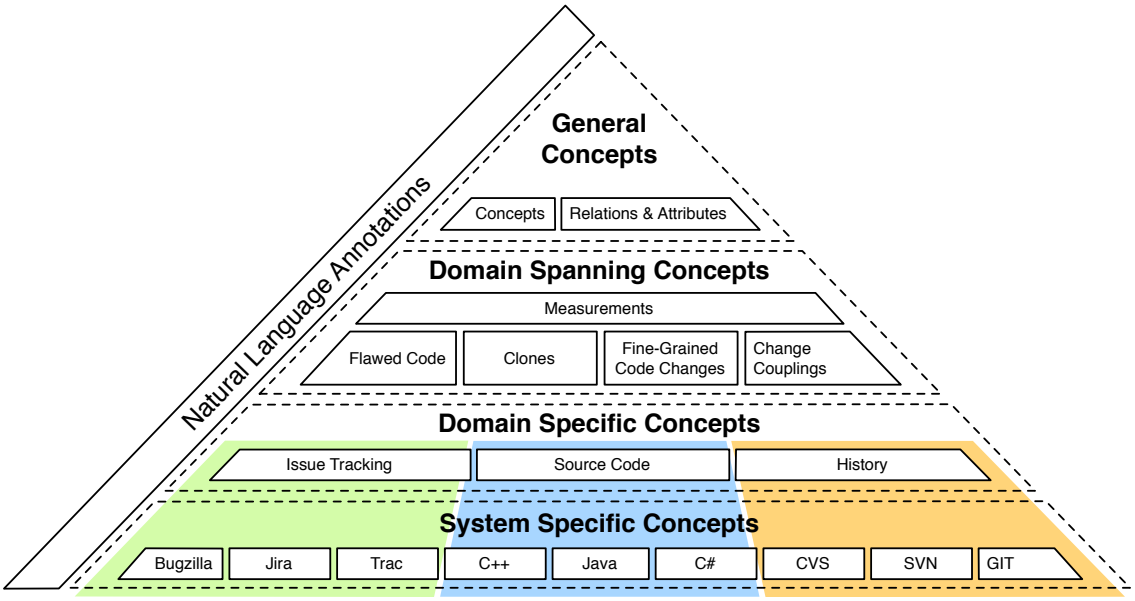

Figure 4.1: The Software Evolution Ontology Pyramid

Figure 4.1 presents an overview of the different layers of *SEON*. The most distinguishing feature is, compared to other ontologies related to the domain of software evolution, the strict organization into different levels of abstraction. In the following, we explain each of the layers that comprise our pyramid of ontologies. We focus on a few examples but do not provide a detailed description for every concept defined in *SEON* in this paper. Instead, we explain the general structure of our ontology pyramid and the rationale behind its design. Interested readers are invited to browse our OWL definitions online.[8] At the end of this section, we give an example on how the different layers can be used in conjunction with each other to describe knowledge in a concrete analysis scenario, namely the analysis of the evolution of code clones in a software system.

4.4.1 General Concepts

The pyramidion, *i.e.*, the top layer, is comprised of domain-independent or general concepts, the attributes that describe them, and the relations between the concepts.

Concepts are modeled by OWL classes. Instances of classes are OWL individuals. OWL datatype properties represent attributes, and OWL object properties the relations

[8] http://www.se-on.org/ontologies/

between concepts. The first ones link individuals to data values, whereas the latter ones link individuals to individuals. To better differentiate terms, we underline OWL classes in this section. A dotted underline denotes individuals and a dashed underline is used for properties.

Classes in the top-layer relate to concepts omnipresent in software evolution. Examples are Activity, Stakeholder, or File. We also defined a set of datatype properties for generic attributes, such as hasSize or createdOn. They are domain-independent; files, program execution stack traces, but also project teams have a size. Similarly, requirement documents, bug reports, or mailing list entries are attributed a creation date.

SEON also defines a more extensive set of domain-independent object properties. These properties are fundamental to many applications, as relations between "things" are paramount for most analyses in software evolution. On this level of abstraction, there is for example the concept of authorship, as any artifact in software evolution has one or several authors, denoted by the object property hasAuthor. Our ontology also has an object property called dependsOn that generalizes many different relations in the software evolution domain.[9] Specializations of dependsOn therefore can range from other domain-independent properties, such as hierarchical relationships (*i.e.*, a *parent-child* relationship), to more domain-specific ones, *e.g.*, dependencies between requirements or static source code dependencies. Such domain-specific properties, however, are specified in lower layers of *SEON*, as sub-properties of higher-level ones. Another domain-independent object property defines the abstract notion of similarity between two individuals. The concept of similarity, again, is universal. It applies to source code (a.k.a. "code clones"), as well as to issues (a.k.a. "bug duplicates") and many other artifacts. What "similar" actually means in a specific case, however, is then up to the fact extractors to decide when they instantiate *SEON* models.

What is the benefit of having defined the abstractions described above? First, we as human beings are comfortable with thinking in categories—this capability develops as early as within the first half year of our lives [HS04]. Categorization and taxonomizing things help us to understand the complex domain of software evolution. Second, as we will describe in the remainder of this paper, such abstractions enable us to build flexible, largely domain-independent tools to support many different facets of software evolution activities.

[9] The concept of inheritance in OWL goes further than in object-orientation. Not only OWL classes can inherit from other classes, but also OWL object and data properties can inherit from other properties.

4.4.2 Domain-spanning Concepts

The second-highest layer of *SEON* defines domain-spanning concepts. These concepts are less abstract than the general concepts. They describe knowledge that spans a limited number of subdomains, *e.g.*, version control systems and source code in the case of our change coupling ontology. Change couplings describe implicit relationships between two or more software artifacts that frequently change together during evolution [BKPS97, GHJ98]. Other ontologies related to the version history of program code cover fine-grained source code changes and code clones. The ontology for fine-grained source code changes describes program modifications not only on a file level but also down to the statement level. It is based on the CHANGEDISTILLER meta-model of change types [FWPG07]. The code clone ontology is able to describe duplicated code and how it evolves over time. Similarly to the code clone ontology, our ontology about flawed code is concerned with quality attributes of source code. The ontology represents knowledge distilled from issue trackers and version control systems. It describes the bug history of files or modules, but also of individual classes or even methods in object-oriented programs. Furthermore, it covers *Design Disharmonies* [LM05] or, in other words, formalized design shortcomings found in source code, *e.g.*, Brain Classes, Feature Envy, Shotgun Surgery, etc.

Another important concept is that of a Measurement. A sophisticated ontology for software measurement has been presented by Bertoa *et al.* in [BVG06]. *SEON* adapts some of the most important concepts identified by these authors, but we weigh simplicity over completeness by leaving out those that have not played a crucial role in our recent analyses.

A measurement is the act of measuring certain attributes of a software artifact or process; a Measure, or metric, is the approach taken to perform a measurement. Measures have a Unit, such as *number of bugs per line of code*. Measured values are expressed on a Scale, *e.g.*, an ordinal or nominal scale. Information about units and scales can be used to perform conversions, for example, to compare the results of different measurements. While the abstract concepts are defined in the pyramidion, many primitive measures are domain-specific. Still we consider measurements to belong mainly to the layer of domain-spanning concepts. Primitive measures, such as *number of lines of code* and *number of closed bugs*, on their own are not very meaningful and need to be put into relation in order to derive a meaningful assessment of a software system's health state. The most effective measurements therefore are based on derived measures [LM05]; they present an aggregation of values from different subdomains. The *number of bugs per class* is

computed from values originating from the source code and the issue tracker, and the *level of class ownership* is derived from source code and commits to a version control system.

In summary, *SEON*'s layer of domain-spanning concepts describes software evolution knowledge on the level of analyses and results, whereas the remaining two layers describe raw data, *i.e.*, artifacts and meta-data directly retrieved from repositories.

4.4.3 Domain-specific Concepts

The third layer is divided into different domains corresponding to important facets of the software evolution process, that is, among others, issue and version management. It includes a taxonomy for source code artifacts encountered in object-oriented programming. While the concepts defined in this layer are specific to a domain, they are independent of technology, vendor, and version. Each domain captures the commonalities shared among the many different issue trackers, object-oriented programming languages, or version control systems.

The majority of issue trackers are organized around Issues that can be divided into Bugs, FeatureRequests, and Improvements. Issues are reportedBy someone and assigned to a developer for fixing them. Object-oriented programming languages usually consist of Classes organized in some kind of Namespaces. Classes declare Members—Methods and Fields—and they can inherit from other classes. Developers modify files in resolving issues and commit them to a version control system resulting in a new Revision for these files. They organize their repository with respect to development streams into Branches and prepare from time to time a Release of the system under development. All these concepts—and many more—are formally defined in *SEON*. These definitions build a taxonomy that can be shared among researchers and practitioners, but also among machines.

Concepts do not necessarily need to be present in all of the systems that are abstracted by the domain-specific layer. The concept of, *e.g.*, Mixins does not exist in Java but in other languages, such as Scala and Smalltalk. Defining this concept nonetheless is perfectly valid, as it is a common concept in object orientation. There will simply be no instances of such concepts if *SEON* is used to describe a software system written in Java or any other language that does not support them.

While devising the layer of domain-specific concepts, we maintained a bird's-eye view on commonly used technologies that are conceptually related, yet very different in implementation. Our goal was to distill some of the essentials of software evolution into a set of meta-models. These meta-models, however, are not static. They are destined to

evolve, as the body of software engineering knowledge grows.

4.4.4 System-specific Concepts

Whereas the third layer describes domain-specific concepts that apply to families of systems, the bottom layer defines system-specific concepts. It extends the knowledge of the upper layers by concepts unique to certain programming languages, vendors, versions, or specific tool implementations. We aim to keep this layer as thin as possible while capturing relevant information beneficial for analyzing specific facets of the evolution of concrete programs. For some systems, we have barely seen the need to define specific concepts, without loosing crucial information. Other systems differ significantly from the baseline and require more system-specific knowledge.

One example for system-specifics is the severity of issues. While most modern issue trackers know the concept of severity to classify an issue, their concrete implementations vary quite substantially. The different levels of severity, as well as their naming, depends very much on the particular issue tracker and, in some cases, even on how it is configured by development teams. Still, the information is valuable, *e.g.*, as input for machine learning algorithms when experimenting with automated bug triaging approaches [GPG10]. Therefore we defined Severity in the layer of domain-specific concepts, but the individuals that represent the different levels of severity are covered in system-specific ontologies. System-specific parsers then extract this information and link individuals of Issue to the corresponding individuals of Severity.

4.4.5 Natural Language Annotations

The Semantic Web was not primarily devised for human beings consuming information. Instead its conception is that machines become capable of processing the knowledge of humans and there is usually additional effort of knowledge engineers needed to encode it in an adequate format.

Despite this machine-centric design, there are many occasions where humans need to interface with Semantic Web data. Therefore, we added a layer of natural language annotations to *SEON*. These annotations provide human-readable labels for all classes and properties. For individuals, we use RDF Schema labels (*rdfs:label*).

In particular, we defined the following custom annotations as subclasses of the OWL

AnnotationProperty.[10] The most important three annotations in the natural language layer are:

- **phrase-s** adds singular synonyms to OWL classes and properties.
- **phrase-p** adds plural synonyms to OWL classes and properties.
- **explanation:** adds a human-readable description to OWL classes and properties

The encoding of the grammatical number of a synonym (*phrase-s* vs. *phrase-p* annotation) is important in order to correctly translate statements from OWL to natural language. The *explanation* annotation is very similar to the RDF Schema comment annotation (*rdfs:comment*) defined by the W3C, except that our annotation is explicitly meant to be shown in user interfaces to end-users (*e.g.*, in tooltips), whereas *rdfs:comment* is also often used to document OWL classes and properties for knowledge engineers.

In Figure 4.2, we show an excerpt of an RDF graph as an example of how we annotate our SEON ontologies with natural language. For the concept Developer, we added multiple natural language representations, in particular the nouns *Author(s)*, *Developer(s)*, and *Programmer(s)*. The annotations from its super-concept Stakeholder—*Stakeholder(s)*, *Person(s)*, and *People*—also apply to Developer. Same applies for properties, where for example changes is annotated with the verbs *change(s)*, *modify*, *modifies*, and *edit(s)*.

In contrast to OWL classes and properties, where the annotations are encoded directly in *SEON*, fact extractors have to generate meaningful *rdfs:label* values for individuals. In most cases, this process is straightforward: for Java classes, fields, and methods, the Java identifier is taken, whereas for bug reports, the issue-key provided by the issue tracker (*e.g.*, "IVY-123" for the issue #123 of the Apache Ivy project) serves as label.

Both, the annotations and *rdfs:labels* are key to the query approach that we discuss in Section 4.5.2. When entering queries, the nouns and verbs are used to provide guidance in composing questions, such as *"Which Programmer modifies the method foo()?"* or *"What methods call bar()?"*. The natural language layer of *SEON* also enhances some of the Web front-ends of the software analysis services presented in Section 4.5.1. The annotations are used to automate the generation of simple human-readable reports, *e.g.*, "Michael Würsch commits Revisions 1-100." or "The class DBAccess has changed 50 times.".

[10] `http://www.w3.org/2002/07/owl#AnnotationProperty`

4.4.6 Our Knowledge Engineering Process

Choosing which concepts should be included in an ontology in general, and assigning concepts to a layer of *SEON* in particular, is not always straight-forward. In the following we therefore briefly sketch the informal ontology design process used for *SEON*, which is illustrated in Figure 4.3.

Knowledge engineers often start from an abstract high-level view when they identify and describe the important concepts in a domain under discourse. Then these concepts are iteratively validated and refined against the reality. In contrast to this top-down approach, we follow a more data-driven, bottom-up approach. At the beginning of the conception phase of a new software evolution support tool or data importer, we quickly model the important concepts of its domain, while neglecting those concepts that are not of immediate use for our purpose. For each important concept, we check whether it is already represented in one of *SEON* upper layers, *e.g.*, the domain-specific layer, and re-use the existing concepts whenever possible. If the concept is not yet defined, we first stage the concept in a system-specific ontology for the specific system. Additionally, we check whether we have already defined similar concepts in other system-specific ontologies and, if so, queue them for consolidation. We usually post-pone the consolidation step until we reached a sufficient understanding of the problem domain—system-specific ontologies therefore act like an incubator to new concepts.

When we model, for example, the concepts of the two programming languages Java and C++, we first create two distinct system-specific ontologies. Then we compare the results and move the commonalities, such as Class, Field, Method, extends, invokes, etc., to *SEON*'s domain-specific layer. The concepts that apply only to C++, such as Struct, Function Pointer, Header File, and the Java-exclusive concepts, *e.g.*, Interface, Annotation, and Inner (Anonymous) Class, remain in the respective system-specific ontologies. Pervasive concepts, *i.e.*, those that apply to multiple domains, for example File, are promoted from the domain-specific to the domain-spanning—or even to the general layer of *SEON*.

4.4.7 An Example Scenario: Clone Evolution

Code clone detection in source code has been a lively field of research for many years now and it is generally accepted that duplicated code violates the *Don't Repeat Yourself (DRY)* principle [HT99], which can lead to software that is harder to maintain. An interesting aspect of code duplication is how clones evolve over time. This was, for

example, investigated by Kim *et al.* in [KSNM05].

Now consider the following scenario, where a researcher decides to carry out a similar study to the one presented by Kim *et al.* In particular, the researcher wants to find out whether the number and size of duplicated fragments change over the lifetime of a Java program. We assume that a clone detector was already selected; scripts to check-out every version of the source code files from an SVN repository have been developed. What is left, is to devise a tool that runs the clone detector on the data to perform the analysis. For that, the researcher needs to decide what meta-model should be used to represent the data under analysis, as well as the results of the analysis.

SEON provides all the necessary means to describe such knowledge. In the following, we briefly discuss how the relevant concepts and their relations are distributed over the four layers of our ontology pyramid. The OWL classes and object properties for the scenario are illustrated in Figure 4.4. The illustration omits datatype properties for the sake of simplicity.

The core concept for this analysis is Clone. A clone belongsTo a CloneClass of duplicated fragments that are similar in syntax or semantics. While the concepts of our clone ontology might not suffice to represent all possible variants of clone analyses, it is straightforward to extend the existing ones. For example, one could specialize the concept Clone with different types of clones, such as SemanticClone or SyntacticClone to provide further classification. Or, additional object properties could link clones to issues for investigations on whether duplication leads to more bugs, and so on.

A Committer introduces a clone when she commits a new Version of a VersionedFile to the SVN repository. Committers are Developers that can check-in modifications. They are one of the many Stakeholders associated with the development process. Versioned files are Files managed by a version control system. Files are among the Artifacts that are produced when software is created. Clones occur in a particular CodeEntity, such as in a ComplexType (*i.e.*, a class, interface, enum, etc.), a Method, etc. The size of such a piece of code, as well as the size of a clone, can be assessed by a Measurement. An adequate Measure for that is the number of lines of code, LOC.

The OWL classes Cloning and Commit are special cases: in principle, the relationship between clones and committers is already sufficiently stated by the object property introduces. However, in some cases, we also want to express that the introduction of a clone is an Activity with a certain time stamp and carried out by a particular stakeholder. There are two ways to do that. The first is reification, which allows for statements about statements. The second is to define an association class. Since reification has not been

```
Cloning(?cloning), doneBy(?cloning, ?committer),
manifestsIn(?cloning, ?clone) → introduces(?committer, ?clone)
```

Listing 4.1: An Example for a SWRL rule defined by SEON

widely adopted in the Semantic Web, we decided for the second variant and defined the OWL class Cloning to represent the introduction of a clone. A clone introduction is doneBy a committer and manifestsIn a new clone. A similar case is that of a Commit. It is also an activity that a committer performs and which adds a new version to a file. This apparent redundancy in the ontology definition allows us to support a wider range of applications. The query approach discussed in Section 4.5.2 works better with triples, such as *"Committer$_A$ commits Version$_B$"*, that are close to the subject-predicate-objected sentence structure in English. On the other hand, the tool presented in Section 4.5.3 explicitly queries for activities to generate, *e.g.*, timeline views. Fact extractors do not necessarily need to create both, an individual of Cloning and the statement *"Committer$_X$ introduces Clone$_Y$"*. In many cases, we defined rules in the Semantic Web Rule Language (SWRL) [HPSB$^+$04], similar to the one in Listing 4.1. The rule states that, if some cloning has been done by a committer, and this cloning manifested in a clone, then the committer has introduced a new clone. With a reasoner, we can then automatically infer the missing triples for particular cases.

Notable in Figure 4.4 is also the OWL class Visibility. In most object-oriented programming languages, there exists an information-hiding mechanism to control the access of parts of the code. In Java, there are the visibility modifiers public, default, protected, and private that apply to types and their members. The actual instances of the visibility modifiers are defined in a system-specific (**Java**) ontology because there are quite significant differences in the meaning of such modifiers depending on the programming language used. The visibility concept, however, belongs to the domain-specific layer together with the other abstractions of **Code**. The layer also contains the predefined LOC individual, because the measure is clearly associated with program code. In our analysis scenario, there are no domain-spanning measures needed. The **History** ontology is located at the same level of abstraction as the **Code**. Currently, there are no system-specific extensions to it. The **Clones** ontology is domain-spanning—it relates to the **Code**, as well as to the **History**. The general concepts layer then provides abstractions for various concepts used in the lower layers.

Coming back to our initial clone evolution analysis scenario, we conclude that *SEON*

```
SELECT ?clone ?size ?version
WHERE
  { ?code rdf:type seon:CodeEntity
    ?clone rdf:type seon:Clone ;
             seon:occursIn ?code
    ?version rdf:type seon:Version ;
             seon:contains ?code
    ?measurement rdf:type seon:Measurement ;
             seon:with seon:LOC ;
             seon:hasValue ?size ;
             seon:measures ?clone }
```

Listing 4.2: SPARQL query returning Clones incl. size and version they appear in

provides the concepts necessary to support it. Once the ontology has been populated by a fact extractor, a concise SPARQL query can be issued to retrieve all clones, their size, and the versions they occur in. The query is given in Listing 4.2. Note that we have left out the prefix definition part of the query: the prefix *rdf* refers to `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, whereas we assume that the *seon* prefix stands for `http://se-on.org/ontologies/`. In reality, each of the different layers of *SEON* has its own prefix/namespace.

4.5 Applications powered by SEON

In the following, we describe three different applications that work with *SEON* as their semantic backbone. The first one is our software evolution analysis web service platform *SOFAS*; the second one is HAWKSHAW, a natural language interface for answering program comprehension questions; and the third application is a recommender system called Semantic Visualization Broker (SVB). SVB analyzes the semantics of a given set of data and comes up with a list of visualizations that could be helpful to gain a deeper understanding of the software system under analysis. We have fully implemented the three approaches in proof-of-concept tools. *SOFAS* and HAWKSHAW are even available for download on the *SEON* website.

4.5.1 Software Analysis Services

Mining Software Repositories has been an active field of research for many years, and various analysis techniques have been proposed, based on the idea that software engineers can learn from the development history of programs.

No matter whether these approaches are concerned with code analysis, code duplication, bug prediction, or any of the other repository-based analyses, many of them have in common that researchers had to build data extractors for version control repositories, issue trackers, mailing lists, and so on. While these efforts share many similarities, synergies are hard to exploit as many tools were designed to work stand-alone. The outcome is a diversity of platforms, similar, yet incompatible meta-models, and tool-specific input and output formats.

To overcome these challenges, we have devised *SOFAS*[11] (SOFtware Analysis Services), which we presented in [GG11]. *SOFAS* allows for a simple yet effective provisioning and use of software analyses based upon the principles of Representational State Transfer (REST, as introduced by Fielding in [Fie00]) around resources on the Web.

An overview on the architecture of *SOFAS* is given in Figure 4.5. The architecture is made up by three main constituents: *Software Analysis Web Services*, a *Software Analysis Broker*, and *Software Analysis Ontologies* being part of *SEON*. The software analysis web services "wrap" already existing analysis tools by exposing their functionalities and data through standard RESTful web service interfaces. The broker acts as the services manager and the interface between the services and the users. It contains a catalog of all the registered analysis services with respect to a specific software analysis taxonomy. As such, the domain of analysis services is described in a semantical way enabling users to browse and search for their analysis service of interest. *SEON* defines and represents the data consumed and produced by the different services.

REST provides us a truly uniform interface to describe all the analysis services in the *SOFAS* architecture, the structure of their input and output, and how to invoke them at a syntactic level. However, there is no way to programmatically know what a service actually offers and what the data means that it consumes and produces. Ontologies in general, and *SEON* in particular, help tackling both problems by providing meaningful service descriptions and data representation.

The Semantic Web leverages *SOFAS* in multiple ways. First, every resource gets a de-referenceable URI assigned. URIs align well with the REST principles and allow one service to hand-over artifacts to another one in a straight-forward manner. Next, the formal data semantics achieved with *SEON* helps in clearly specifying the input expected, as well as the output generated by the services, which increases interoperability and simplifies reuse of processing results. This is achieved by slightly expanding the Web Application Description Language (WADL) [Had09] with annotations inspired by SAWSDL (Semantic

[11] *SOFAS* is available online at http://se-on.org/sofas/

Annotations for WSDL) [FL07]. With them, the input and output of the services can be declared as being described by *SEON*. Last but not least, the footprint of the information exchanged by the services can be reduced by incorporating a reasoner. Only a limited set of triples then needs to be passed along by the sender and reasoning can be done by the receiver to add additional triples, if needed.

4.5.2 Supporting Developers with Natural Language

In [WGRG10] we presented a framework for software engineers to answer common program comprehension questions with *guided-input natural language* queries, for example those questions presented by Silito *et al.* in [SMV06]. The framework is called HAWKSHAW[12] and has been implemented as a set of plug-ins for the Eclipse IDE. Figure 4.6 shows a screenshot of our query interface in action. In the example, a user has already started to compose a query. Three words have been typed in so far, *"What Method invokes"*, and the drop-down menu presents the full list of methods that can be entered to complete the query.

The HAWKSHAW approach follows a method coined *Conceptual Authoring* or WYSIWYM (What You See Is What You Meant) by Hallet *et al.* in [HSP07] and Power *et al.* in [PSE98]. This means that, for composing queries, all editing operations are defined directly on an underlying logical representation, in our case *SEON*. However, the users do not need to know the underlying formalism because they are only exposed to a natural language representation of the ontology.

We use a multi-level grammar consisting of a static part that defines basic sentence structures and phrases for English questions, and a dynamic part that is generated when an ontology is loaded [BKKK06]. The static part needs to be defined manually and, additionally, contains information on how to translate the user input into SPARQL. We generate the dynamic part from labels of the individuals, from the identifiers of the classes and properties, as well as from the *SEON* natural language annotations (see Section 4.4).

The static grammar basically defines a stub. In the example given above, the grammar describes that, after one of the interrogative determiners "What" or "Which", the subject of the sentence needs to follow. The subject needs to be an OWL class defined by *SEON*. Further, the verb of the sentence has to be an object property that fits the subject, *i.e.*, the

[12]Our tool is named after Hawkshaw the Detective, a comic strip popular in the first half of the 20th century. *Hawkshaw* meant a detective in the slang of that time. The tool HAWKSHAW is available for download at http://se-on.org/hawkshaw/

object property has the class in its domain that has been selected as the sentence's subject. Object properties not fulfilling this constraint will not be presented to the user. Similarly the object of the sentence is an individual of a class in the ontology. The individual's class has to comply to the range specified for the object property, otherwise it will not be shown either. The stub provided by the static grammar then looks as follows: "What <class> <object-property> <individual> ?"

The dynamic part of the grammar provides the replacements for the placeholders in the stub (denoted by $< >$). These replacements are presented to the user. Consider *"What Method*[1] *invokes*[2] *charge()*[3] *?"*. In this query, ([1]) is a label for the OWL class *JavaMethod*, ([2]) comes from the object property *invokesMethod*, and ([3]) from a human-readable label for one of the OWL individuals that have the class *JavaMethod*.

The utilization of the *SEON* ontologies for driving HAWKSHAW yields several major benefits: Ontologies are described in terms of triples of *subject*, *predicate*, and *object*. This structure strongly resembles how humans talk about things and can be easily transformed into natural language sentences. A surprisingly small set of static grammar rules allows for a variety of different queries.

Properties in OWL are a binary relation that can be restricted by specifying *domain* and *range*. In triples this means that the domain restricts the possible values of the subject and the range restricts the values of the object. For our query approach, this information can be exploited to filter the verbs that can follow a given subject, or the objects that can follow a given verb. For example the question *"Which developer is assigned to issue #133?"* makes sense, whereas *"What field invokes class A?"* does not.

We employ the Pellet reasoner [SPG+07] to infer specializations or generalizations. When we ask for, *e.g.*, *"What persons are contributing to project X?"*, we are not only interested in a list of direct instances of the concept *Person*, but also in specializations, such as *Developers*, *Testers*, etc. Similarly, whenever we know that developers *create* or *change* an artifact, we also want to generalize that they are *contributing* to the project. Reasoners greatly simplify data extraction, as they reduce the amount of explicit information that we need to state in our models.

4.5.3 Semantic Visualization Broker

The third application presented in this paper addresses the hardly known capabilities of software visualizations. The Semantic Visualization Broker (SVB) is essentially a recommender tool that suggests to the user suitable visualizations for a given set of data.

The data can originate from the results of a query composed with the HAWKSHAW approach (Section 4.5.2), but also from a *SOFAS* analysis workflow (Section 4.5.1), or virtually any other source of RDF/OWL data.

Visualization plug-ins can register themselves with the SVB and specify the semantics of the data they can handle. The SVB expects as input a knowledge base and a result set. The result set should consist of the information a user asked for, whereas the knowledge base provides the context, in case that the SVB or a visualization has to query for additional data. The SVB then invokes a reasoner to infer abstractions from the result set and compares the outcome with the registration that the visualization plug-ins have provided. Any matches are presented to the user. The user can then select one or several recommendations from the list and the SVB will invoke and configure the visualizations automatically with the input data.

When the SVB receives a set of individuals as result set, it will query the knowledge base for their data properties and for object properties that link those individuals together. We currently support four different scenarios, which we describe in the following. An overview on the implemented visualization types is given in Figure 4.7.

Hierarchies. If the SVB detects a hierarchical relationship between the individuals in the result set, it will recommend a simple tree-like widget (which has been omitted from Figure 4.7—it is similar to the widgets well-known from file system explorers) and a tree map visualization. If the selected individuals have a size measurement assigned (*e.g.*, for files the *lines of code* metric), the SVB will configure the tree map to incorporate the size of each individual to calculate the layout.

Measurements. If more than one individual has measurements assigned, then the SVB recommends a visualization based on Radar Charts. Each axis of the chart represents a certain type of measure. The number of axes that are displayed is limited; whenever more measures are available, some of them are chosen randomly and the user is given the possibility to reconfigure the selection. If measurements are available for more than one version of the individuals (*e.g.*, for files under version control), then each axis will display multiple entries.

Activities. In the case that most of the individuals represent an activity with a timestamp assigned, the SVB will automatically come up with a scrollable timeline-like visualization.

Miscellaneous data. As a fallback, if none of the cases above apply, the broker will suggest a simple graph-based explorer that displays individuals and data values as nodes and properties as edges. Unless the properties are defined as being *symmetric*, the

corresponding edges will be directed.

Labels displayed in each of the visualizations are derived either from the RDF Schema labels or from the natural language annotations of *SEON*. The clear, machine-processable semantics of the data enable the SVB to make educated guesses on what visualizations may be appropriate. The power of a reasoner allows us to specify the concepts and relations supported by a visualization in a very generic way—the reasoner will automatically infer a hierarchical relationship from a set of triples containing, *"$Class_A$ declaresMethod $Method_B$"* and propose a tree-based visualization consequently.

The SVB offers quite some potential for enhancements. For example, we will explore the range of visualizations it can support and to what extent it is generalizable to non-visual applications.

4.6 Related Work

In this section, we briefly sketch existing work involving ontologies in software engineering. We refrain from discussing publications that are only related to the approaches presented in Section 4.5, but not particularly to the Semantic Web and ontologies. Related work in the context of software analysis services was already given in [GG11], whereas research in the area of program comprehension and developer support has been discussed extensively in [WGRG10].

A general overview of applications of ontologies in software engineering has been given in [GL02, HS06, UJ96]. All of these publications promoted the theoretical benefits offered by different characteristics of ontologies, such as explicit semantics and taxonomy, lack of polysemy, ease of communication and automatic data exchange between distinct tools, and computational inference. In the following, we elaborate on how ontologies were applied to advance particular fields of research in software engineering. To the best of our knowledge, *SEON* is the only approach that describes software evolution data on multiple abstraction layers. Another unique selling proposition of our family of ontologies is that they were validated in three very distinct scenarios (*cf.* Section 4.5), whereas most other ontologies were deployed only in a rather specific environment.

4.6.1 Ontologies for Software Artifacts

Different approaches to establish taxonomies for software engineering by means of ontologies have been presented recently.

Hyland-Wood *et al.* [HWCK06] proposed an OWL ontology of software engineering concepts, including classes, tests, metrics, and requirements. Bertoa *et al.* focused on software measurement [BVG06]. Their software measurement ontology influenced the respective concepts of *SEON*. Bertoa *et al.*'s set of measurement concepts is more complete, whereas our ontology focuses on simplicity.

Oberle *et al.* recognized that *the domain of software is a primary candidate for being formalized in an ontology* [OGS09], being both, sufficiently complex and reasonably stable in paradigms and aspects. Consequently, a reference ontology for software was presented to distinguish fundamental concepts in the domain of software engineering, such as data and software.

These three approaches show some overlap in concepts with our ontologies but they neglected evolutionary aspects, whereas *SEON* explicitly models the development history of software systems, such as versions, releases, bugs, etc.

4.6.2 Ontologies for Software Maintenance

Several approaches relied on ontologies to support software maintenance—be it to describe domain knowledge of developers, source code and documentation to support program comprehension, or to infer bugs based on a set of heuristics.

LaSSIE, presented by Devanbu *et al.* in [DBS91], was an early attempt to integrate multiple views on a software system in a knowledge base. It also provided semantic retrieval through a natural language interface. Frame systems, a conceptual predecessor to the ontologies of the Semantic Web, were used to encode the knowledge. The main goal of LaSSIE was to preserve knowledge of the application domain for maintainers of the software system.

The author of [Wel97] found LaSSIE's source code model too course-grained and not applicable to object-oriented code. Therefore, he augmented abstract syntax trees with semantics. For that DL was used to develop an ontology for software understanding. The ontology, in combination with an inferencer, then enabled automatic detection of side effects in code and path-tracing.

Witte *et al.* [WZR07] used text mining and static code analysis to map documentation

to source code for software maintenance purposes. These mappings were represented in RDF.

Yu *et al.* also represented static source code information by means of an OWL ontology [YZY+08]. They further used the Semantic Web Rule Language (SWRL) [HPSB+04] to describe common bugs found in code. With a rule engine, inference results could be obtained to indicate the presence of bugs.

Our natural language query approach HAWKSHAW described in Section 4.5.2 shares many similarities with LaSSIE but, thanks to *SEON*, potentially covers a broader range of concepts. However, *SEON* does not incorporate application-specific knowledge. The other three approaches described above focus only on source code, whereas we incorporate many different artifacts, stakeholders, and their activities.

4.6.3 Ontologies for Software Reuse

Properties of software components have been represented with ontologies in the past. Such properties ranged from programming languages and source code facts to licenses, software types and application domains. The common goal was to foster reuse by enabling searches in a component database for certain criteria that relate to, *e.g.*, particular requirements.

Happel *et al.* [HKST06] proposed various ontologies to foster software reuse. In their KOntoR approach, they provided background knowledge about software artifacts, such as the programming language used or licensing models. The artifacts, along with their ontology meta-data, were stored in a query-able central repository to facilitate reuse.

The authors of [HKF08] used ontologies to describe software components. They classified software with respect to a hierarchy of software types. An example given in their paper was IBM's DB2, which is a relational database management system (RDBMS); RDBMSs were then considered as a subclass of database managements systems, and so on. The authors additionally defined hierarchies of functionality types (*e.g.*, *importing data* as a special kind of *adding data*) to further describe the features of components. An algorithm was presented to automatically find an optimal component solution for a given set of requirements.

Dietrich *et al.* developed a tool that scans the abstract syntax tree of Java programs and detects design patterns for documentation purposes [DE05]. The design patterns were described in terms of OWL ontologies.

Alnusair and Zhao [AZ11], similar to Hartig *et al.*, used OWL ontologies for component descriptions. They took a three-layered approach for their ontological descriptions: an

ontology representing static source code information, different domain ontologies to conceptualize the domain of each component (*e.g.*, finance or medicine), and an ontology that extended their source code ontology with component-related concepts. The authors supported several kinds of query methods against their component knowledge base: type or signature-based queries, meta-data keyword queries, or pure semantic-based queries.

SEON, in contrast to these four approaches, does neither model software systems at a component level, nor does it represent design patterns. However, in our ontologies, we model other important facets of software that could yield interesting synergies when synthesized with these ontologies for software reuse, for example, to give insights on the maintainability of particular components. This could help software engineers to make even more profound decisions on what components their software systems should be based on.

4.6.4 Ontologies in Search-Driven Software Engineering

The field of search-driven software engineering has produced various code search engines. Some of them simply use OWL/RDF as an internal representation of program code and allow users to issue SPARQL queries against the code base [KRSR10]. Others exploit the possibilities of the Semantic Web further. Durão *et al.*, for example, classified source code according to domains, such as Graphical User Interfaces, I/O, Networking, Security, and so on [aVAdLM08]. The authors then provided a keyword search over the code base, and the results of the queries could be limited to return only matches from a particular domain.

The applications of *SEON* presented in Section 4.5 also make extensive use of the Semantic Web's search facilities, in particular of SPARQL. Source code search, however, is not the main purpose of our applications but rather a means to an end. Nevertheless, it is easily conceivable that we might adopt a code search engine as a *SOFAS* service in the future.

4.6.5 Ontologies in Mining Software Repositories

Several researchers have described software evolution artifacts found in software repositories with OWL ontologies. Their approaches integrated different artifact sources to facilitate common repository mining activities. The flexible RDF data model, automatic semantic mashup technologies, and the powerful search-facilities of the Semantic Web

have proven their use in this context.

Tappolet made a case for incorporating Semantic Web technology in software repositories in [Tap08]. The authors claimed that this would greatly facilitate the handling of distributed and heterogeneous software project data. Tappolet then presented a roadmap towards such semantics-aware software project repositories consisting of three main steps: 1) data representation by means of RDF/OWL ontologies, 2) intra-project repository integration, and finally 3) inter-project repository integration.

Based on these ideas, Kiefer *et al.* presented EvoOnt [KBT07], a software repository data exchange format based on OWL. EvoOnt involved three sub-ontologies: a software ontology model, a bug ontology model, and a version ontology model. The authors used a modified version of SPARQL to detect bad code smells, calculate metrics, and to extract data for visualizing changes in code over time. A reasoner was incorporated to detect orphan methods, *i.e.*, methods never called by any other methods in the system. Tappolet *et al.* recently extended the EvoOnt approach. Several software evolution analysis experiments from previous Mining Software Repositories Workshops were repeated and it was demonstrated by the authors that, if the data used for analysis were available in EvoOnt, then the analyses in 75% of the selected MSR papers could be reduced to one or at most two simple SPARQL queries.

Iqbal *et al.* discussed different scenarios and use cases for Linked Data in software engineering in [IUHT09]. They presented their *Linked Data Driven Software Development* (LD2SD) methodology, which involves transformation of software repository data into the RDF format and then indexing with a semantic indexer. The overall goal was to provide a uniform and central RDF-based access to JIRA bug trackers, Subversion, developer blogs, project mailing lists, etc. Integration between the repositories was achieved with *Semantic Pipes*, an RDF-based mashup technology. The results were finally injected into the DOM of a Web page (*e.g.*, that of a bug tracker) to provide developers with additional, context-related information.

None of these approaches organize their ontologies in consecutive layers of abstractions with clear representational purpose, as we did for *SEON*. Instead, the authors have laid out their ontologies at a particular level of abstraction. For example, while most concepts in EvoOnt can be mapped 1:1 to concepts in *SEON*, it is conceptually situated somewhere between *SEON*'s system- and domain-specific layers and lacks the domain-spanning and general concepts that we have defined.

Despite these limitations, we can envision interesting interactions between our semantics-aware applications and the technologies presented by the other authors. The SPARQL

extension presented by Kiefer *et al.*, for example, adds machine learning algorithms (SPARQL-ML [KBL08]) and similarity joins (iSPARQL [KBS07]) to the Semantic Web. Both extensions could lead to a complete new family of *SOFAS* services or at least simplify the implementation of existing ones. The semantic mashup technology used in LD2SD could further improve the presentation of the analysis results of our services.

4.7 Conclusions

Some decades ago, a team of developers could write industrial-strength software on their own, only with the aid of a simple text editor, a compiler, and perhaps a debugger. The software engineering landscape has changed dramatically since then.

Development teams have grown to dozens, and sometimes even hundreds of people. A plethora of tools have found their way into integrated development environments—without the help of these IDEs, we as programmers can barely imagine to write a single line of code anymore. Software repositories, such as version control systems and bug trackers, foster collaboration and provide means to control and reflect on the development processes.

With the increase in team size and tool support, the amount of data that breaks in on individual developers has grown to a point where it becomes harder and harder for them to grasp implicit relationships among artifacts stored in different locations. Too much time is lost in distinguishing useful information from random noise. In consequence, software engineers are hardly able to fully exploit all their tooling and productivity gains are thus wasted. A new generation of tools is therefore needed—tools that can make use of the semantics of the underlying data to automate tedious processes and filter irrelevant information. The Semantic Web provides a framework to build such tools.

In this paper, we have investigated the research question how software evolution knowledge can be adequately represented by means of ontologies. As an answer to this question, we presented *SEON*, a family of ontologies that describe many different facets of a software's life-cycle. *SEON* is unique in that it is comprised of multiple abstraction layers. Our ontologies provide a shared taxonomy of important software engineering concepts and already have found multiple applications. Three of them were discussed in this paper, and we argued that each application clearly benefits from the use of Semantic Web technologies. *SOFAS*, our software analysis services platform, used *SEON* as a formal description of the input and output of its individual services. Our guided-input natural language approach HAWKSHAW exploited the clear semantics of OWL to translate

program comprehension questions formulated by developers in quasi-natural language to the formal Semantic Web query language SPARQL. This was possible, since the natural language annotation layer of *SEON* bridged the gap between machine-processable and human-understandable knowledge. SVB, our Semantic Visualization Broker, relied on reasoning and explicit relations to automatically infer suitable visualizations for given sets of data. All of these three applications would have been significantly harder to implement without *SEON* and the use of Semantic Web technologies.

We only have started to exploit the potential that the Semantic Web could bring for software evolution support. Other researchers have begun to explore the opportunities and we hope that this paper can encourage even more to do so. A next important step is to consolidate other existing ontologies and to come up with layers of abstraction, similar to what we did with *SEON*. Based on this, software repositories need to be devised that are semantics-aware, *i.e.*, that produce and consume data in the RDF/OWL format, and that expose stable de-referenceable URIs on the Web. When this is achieved, software repositories could ultimately blend into a queryable global information space of interlinked software evolution data.

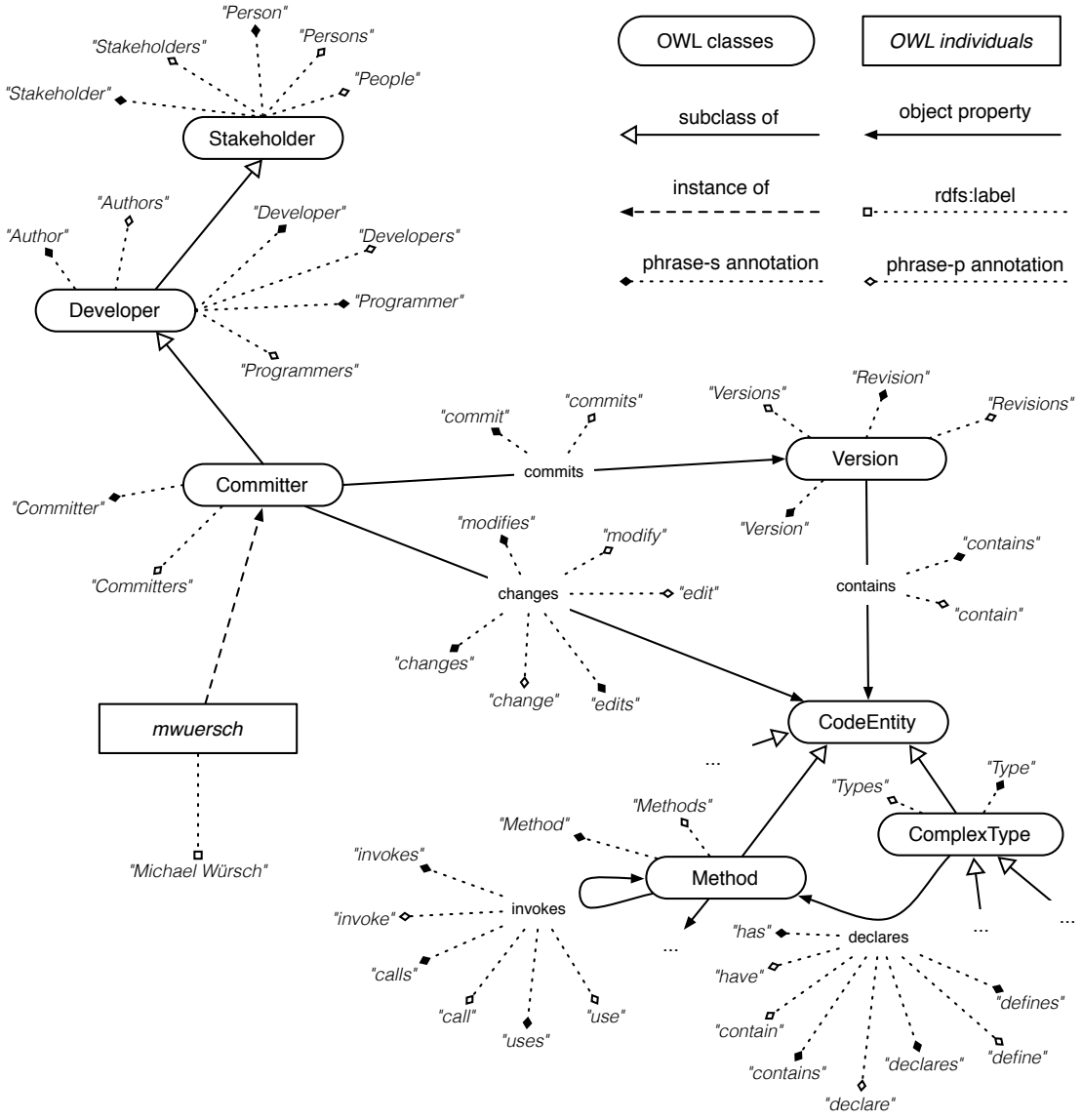

Figure 4.2: RDF Graph with Natural Language Annotations

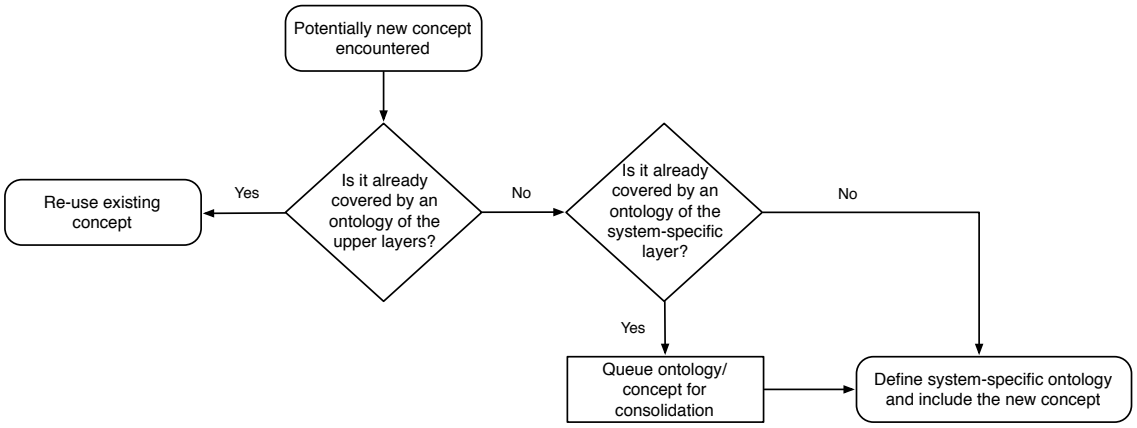

Figure 4.3: Informal Design Process when encountering concepts during the conception of an analysis

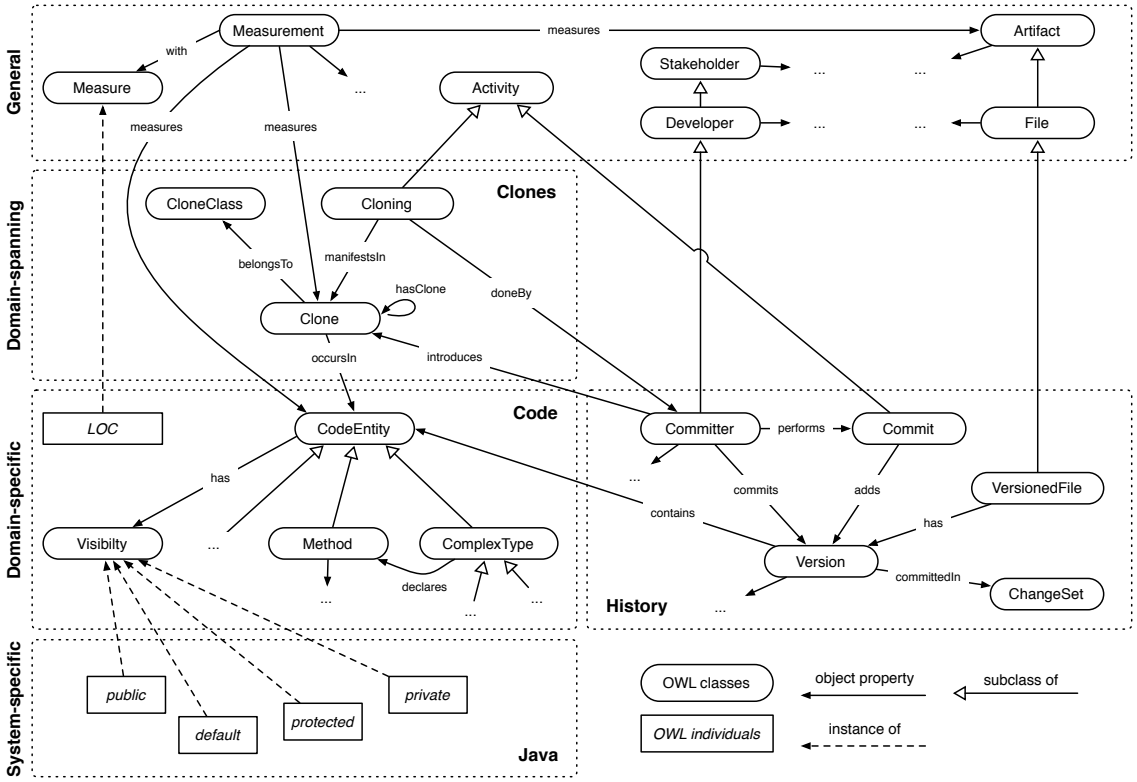

Figure 4.4: The SEON concepts involved in a Clone Evolution Analysis Scenario

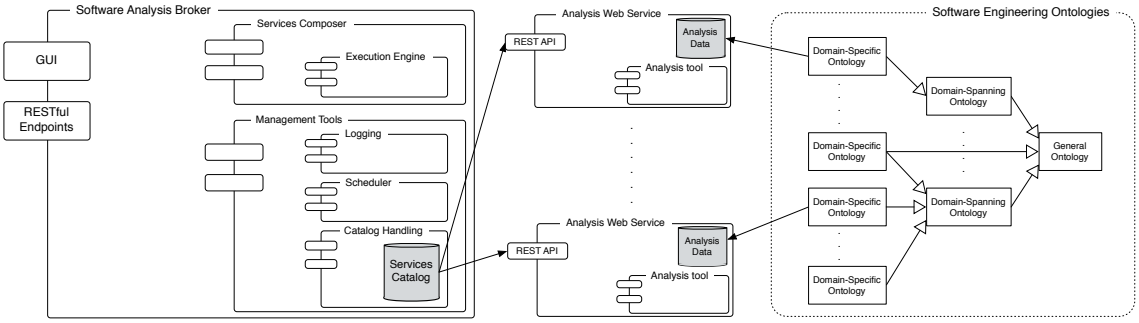

Figure 4.5: The SOFAS Architecture ([GG11])

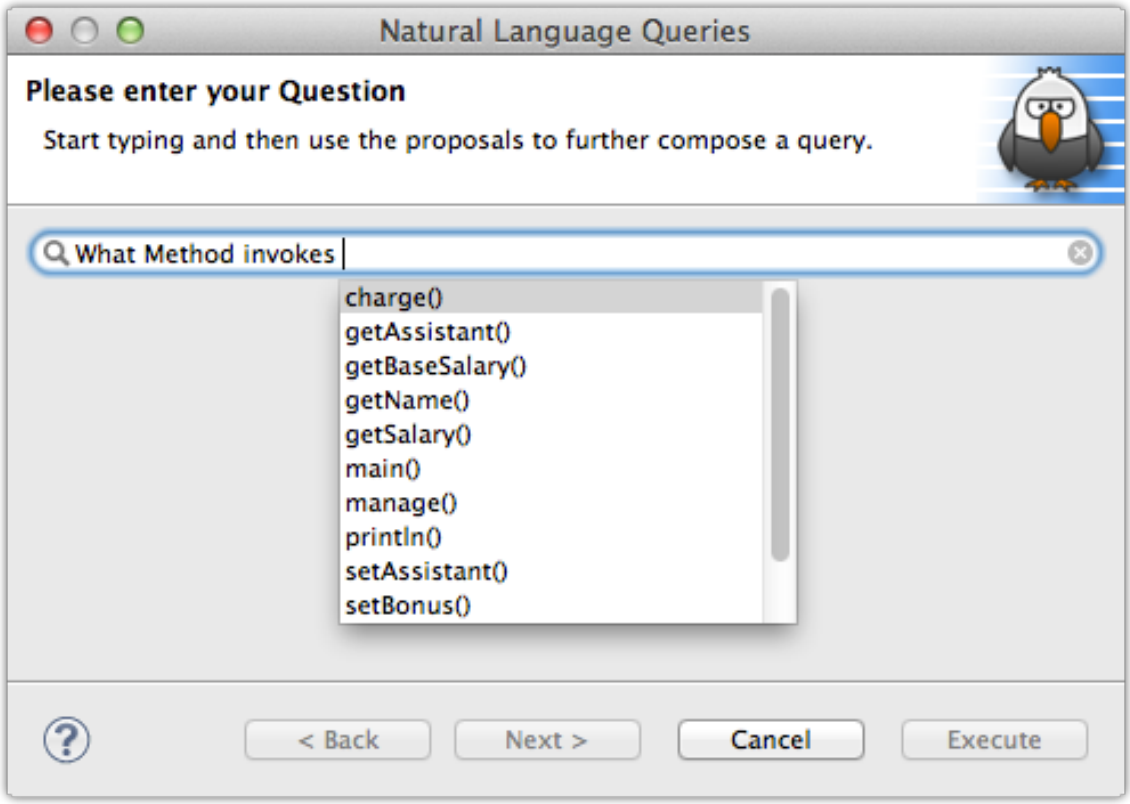

Figure 4.6: The Guided-Input Natural Language Interface powered by SEON

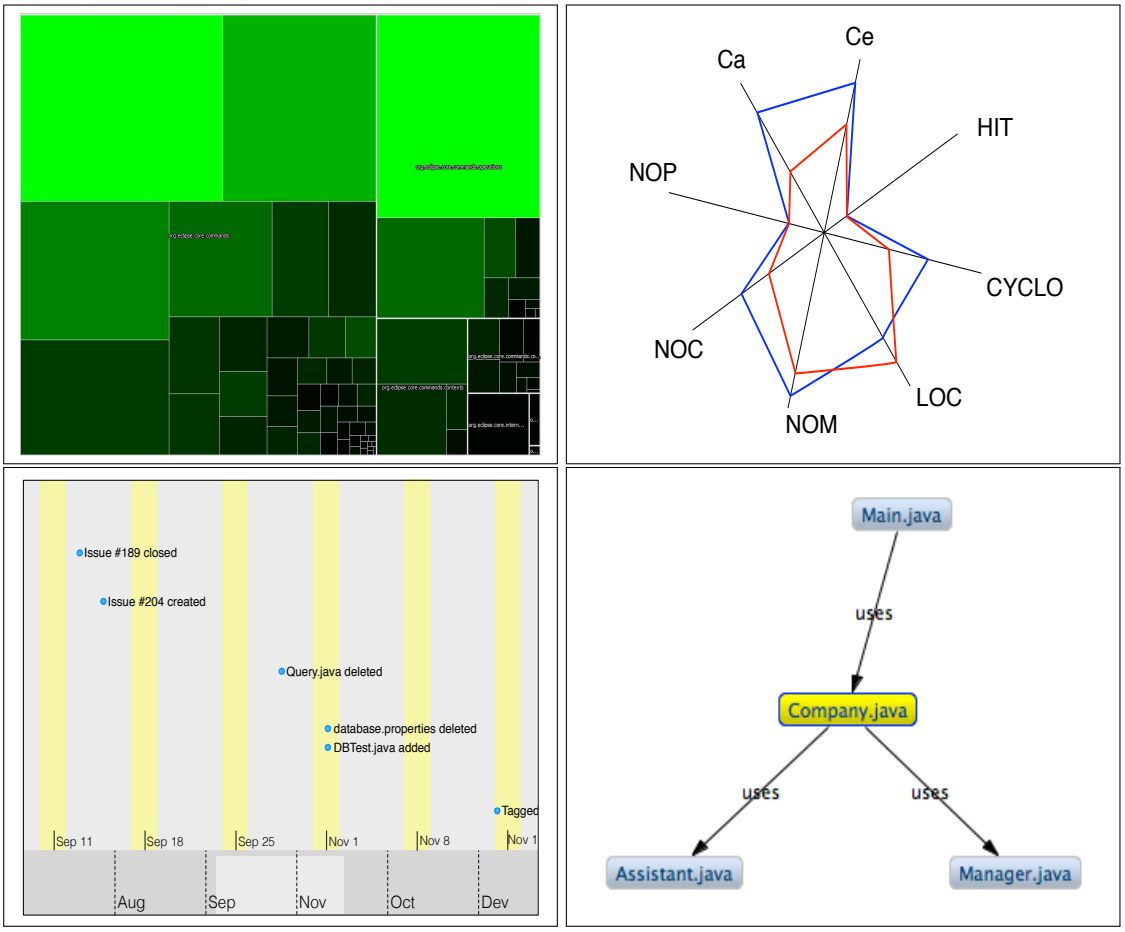

Figure 4.7: The types of visualizations currently supported by the Semantic Visualization Broker: The upper left figure shows a tree map of a Java system, the upper right one shows a radar chart with measurements for two different versions of a Java class, the lower left figure shows a timeline with software evolution activities, and the lower right one shows a simple graph-based explorer displaying the dependencies among four Java classes.

5

Another use of *SEON*

Supporting Developers with Natural Language Queries
Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C. Gall
Proc. Int'l Conference on Software Engineering (ICSE), 2010

DOI: 10.1145/1806799.1806827

THE feature list of modern IDEs is steadily growing and mastering these tools becomes more and more demanding, especially for novice programmers. Despite their remarkable capabilities, IDEs often still cannot directly answer the questions that arise during program comprehension tasks. Instead developers have to map their questions to multiple concrete queries that can be answered only by combining several tools and examining the output of each of them manually to distill an appropriate answer. Existing approaches have in common that they are either limited to a set of predefined, hardcoded questions, or that they require to learn a specific query language only suitable for that limited purpose. We present a framework to query for information about a software system using guided-input natural language resembling plain English. For that, we model data extracted by classical software analysis tools with an OWL ontology and use knowledge processing technologies from the Semantic Web to query it. We use a case study to demonstrate how our framework can be used to answer queries about static source code information for program comprehension purposes.

5.1 Introduction

Program comprehension plays an important role when performing software engineering activities on large bodies of source code. Both industry and research therefore have aimed to integrate various tools into modern integrated development environments (IDEs) to support software engineers in understanding a software system during their daily development and maintenance tasks.

Especially for novice developers, mastering all the powerful features that are delivered by an IDE, such as Eclipse or Visual Studio, requires a great deal of learning effort. When solving a program comprehension task, developers usually have particular questions in mind, such as *"Where is this method called?"* or *"What fields are declared as being of this type?"* [SMV06, SMV08]. Unfortunately, such questions often can not be answered directly using existing functionality offered by IDEs. Instead, as explained by De Alwis and Murphy in [dAM08], developers first need to map these *conceptual queries* to one or several *concrete queries*. Even if a particular conceptual query is directly supported, novice programmers are often not yet aware of the capabilities of their IDE. For example, although experienced developers can easily answer *"Where is this method called"* with Eclipse, newcomers still need to be aware that the *"Find references..."* feature, hidden in a context menu, is what they are looking for.

Existing approaches to enable the integration of different information sources often do not allow developers to formulate ad-hoc queries. Instead, they need to be explicitly configured to enable new queries. On the other hand, query languages, such as CodeQuest [HVdM06] or JQuery [JV03], allow developers to formulate queries about software engineering artifacts. These languages are usually based on a SQL- or Prolog-like syntax and effectively using them requires learning effort. According to Chowdhury, however, *"the most comfortable way for a user to express an information need is as a natural language statement."* [Cho04]. Henninger even suggests that constructing effective natural language queries is as important or more important than the retrieval algorithm used [Hen94].

In this paper, we present a framework that allows software engineers to use *guided-input natural language* strongly resembling plain English to query for information about a software system. For that, we combine software evolution data provided by EVOLIZER, our platform for software evolution analysis, with Semantic Web technologies for knowledge processing. We focus on queries concerning static source code information, such as *"How often is this field accessed?"* or *"What are the subclasses of this class?"*, to demonstrate the potential of our approach; but including more data from various software repositories

and tools is straight-forward, as we will explain in detail in this paper.

The remainder of this paper is structured as follows: In Section 5.2 we give an introduction to the Semantic Web and discuss the knowledge processing technologies that we use throughout the paper. We present our framework to query static source code information with (quasi) natural language in Section 5.3. Section 5.4 provides a case study evaluation of our approach. Section 5.5 discusses existing work related to our approach and Section 5.6 concludes the paper.

5.2 Background

The technologies originally developed for the Semantic Web have been proven useful whenever knowledge has to be processed by machines. In this paper, we exploit them to bridge the gap between more classical software analysis tools and natural language query interfaces.

Tim Berners-Lee *et al.* [BLHL01] define the Semantic Web as *"an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."* Following this definition, this semantic extension enriches the Web with meta-data describing the semantics of the webpages in a computer-processable way. Before the webpages can be described with meta-data accordingly, an *ontology* has to be defined for the domain of discourse. An ontology formally describes the concepts (classes) found in the domain, the relationships between these concepts and the properties used to describe them [Gru93]. For example, in the domain of an online record shop, we define concepts, such as *Composer*, *Album*, and *Track*; the relationships *composed by* and *has track*, and the properties *has play-time* and *has title*. The meta-data description of a CD is then able to give the data a well-defined meaning, using the concepts, relationships, and properties defined in the ontology. In the software engineering domain, we define concepts, such as *User*, *Developer*, *Bug*, *Module*; relationships, such as *reports bug*, *fixes bug*, and *is assigned to bug*. Since the Semantic Web describes this information based on formal semantics, data can be exchanged among two applications that support the same ontology, even if they were not meant to interoperate in the first place.

To bring the vision of the Semantic Web into being, the research community came up with a number of standards, W3C recommendations, development frameworks, APIs, and databases. The Resource Description Framework (RDF) [KC04] is the data-model for representing meta-data in the Semantic Web. The RDF data-model formalizes meta-data

based on *subject – predicate – object* triples, so called RDF statements. RDF triples are used to make a statement about a resource of the real world. A resource can be almost anything: a bug report, a person, a Web page, a CD, a track on a CD, etc. Every resource in RDF is identified by a Uniform Resource Identifier (URI) [BLFM98].

In an RDF statement the subject is the thing (the resource) we want to make a statement about. The predicate defines the kind of information we want to express about the subject. The object defines the value of the predicate. In the RDF data-model information is represented as a graph with the statements as nodes (subject, object) connected by labeled, directed arcs (predicate). The query language SPARQL [PS08] can be used to query such RDF graphs.

RDF is domain-independent in that no assumptions about a particular domain of discourse are made. It is up to the users to define specific ontologies in an ontology definition language, such as the Web Ontology Language (OWL) [De04].

OWL enables the use of description logic (DL) expressions to further describe the relationships between classes and to restrict the use of properties [PSHe04]. For example, two classes can be declared to be disjoint, new classes can be built as the union/intersection of others, or the cardinality of a property can be restricted to define how often a property can be applied to an instance of a class.

In addition to the W3C recommendations, the Semantic Web community developed tools to process RDF meta-data. *Jena*[1] emerged from the *HP Labs Semantic Web Program* and is a Java framework for building applications for the Semantic Web. It provides a programmatic environment for RDF and OWL.

In this paper, we do not contribute directly to the Semantic Web, but we exploit the technologies introduced above to describe and process data. In short, we model software evolution data with an OWL ontology. Then we take, for example, the static source code information that was extracted with our EVOLIZER toolset, and represent it as an RDF graph that is based on this ontology. RDF graphs, in contrast to relations in a relational model, consist of {*subject, predicate, object*}-triples which are close to the natural English sentence structure. This enables the transformation from natural English queries into SPARQL queries which can be issued on the RDF graph. In the remainder of the paper, we describe in detail how the RDF graph is generated and how this knowledge representation is exploited to support software developers.

[1] http://jena.sourceforge.net/

5.3 Approach

Our vision is to provide a convenient and intuitive interface that allows software engineers – and especially newcomers, who are not yet virtuous with their IDE or command line tools, such as grep and awk – to leverage information sources about various aspects of a software system under development. In particular we want to overcome some of the limitations that existing approaches suffer from: We do not want to restrict developers to a set of predefined queries and we do not want them to learn a specific query language for that limited purpose. Instead, we want developers to be able to use a query language that they are already very familiar with: natural language. A natural language interface to an IDE relieves especially novice programmers from the cognitive burden of translating a conceptual query to a concrete one, which might involve navigating through nested or multi-level menus.

In the following, we briefly introduce EVOLIZER, our platform for software evolution analysis. For the sake of this paper, we focus on the infrastructure that is needed to give developers the possibility to query static source code information from within Eclipse. However, we want to emphasize again that our approach can be generalized to many other domains in the software evolution context.

5.3.1 Evolizer

EVOLIZER[2] basically stems from the idea of having a Release History Database (RHDB) [FPG03a] that integrates information originating from various repositories, such as CVS and Bugzilla, in a single database. It facilitates many common preprocessing steps [ZW04] that are necessary when mining such software archives and, in that context, it is comparable with *Kenyon* of Bevan *et al.* [BWKG05] or *eROSE* of Zimmermann *et al.* [ZWDZ05].

Over the years EVOLIZER has advanced from a set of data importers and preprocessors to a platform for various tools that aid developers during their daily maintenance and development tasks. Realized as Eclipse plug-ins, the functionality of EVOLIZER is available at developers' fingertips in a state-of-the-art IDE and incorporates applications that emerged from ideas of the software evolution research community, as well as more classical approaches to support, for example, program understanding. Some of the tools that are built upon EVOLIZER, such as CHANGEDISTILLER [FWPG07, GFP09], follow the original idea of leveraging historical data. Others do not make use of any evolutionary data

[2] http://www.evolizer.org/

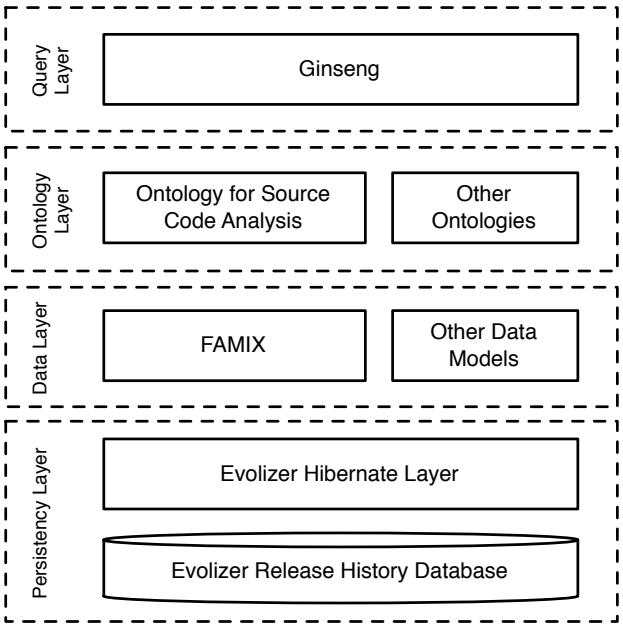

Figure 5.1: The four layers of Evolizer.

at all. Instead, for example in case of DA4JAVA [PGKG08], they analyze source code that is currently under development within the IDE.

Figure 5.1 gives an overview on the four layers of EVOLIZER that are relevant for this paper. The persistency layer gives access to facts about a system in a convenient way and provides application support for other EVOLIZER plug-ins to persist settings, query results and so on. In the following, we provide detailed insight into the other three layers: In Section 5.3.2 we describe the *data layer* consisting of a set of data models and data importers. Section 5.3.3 describes the *ontology layer* that enhances the data layer with formally described data semantics. In Section 5.3.4 we show how existing Semantic Web query technology can be used to query an ontology-based knowledge base with quasi-natural language. Section 5.3.5 sums up how the three layers play together to allow a developer to access facts about her source code in a convenient and intuitive way.

5.3.2 Evolizer Data Layer

EVOLIZER provides a set of data models to represent software evolution data along with adequate importer tools to obtain this data from software project repositories. Extending existing data models and data importers, or adding new ones, is straight-forward. For

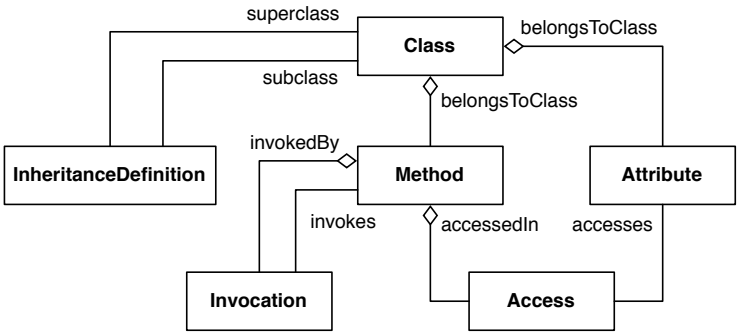

Figure 5.2: Core of the FAMIX model by Tichelaar *et al.*

the approach that we present in this paper, we use a tool that was implemented on top of our platform to perform static source code analysis and store the result in our EVOLIZER RHDB.

To add new data, we first identify the key concepts that we want to analyze (in case of analyzing source code: packages, classes, methods, accesses, invocations, etc.) and create a data model accordingly. We use the plug-in extension facilities of Eclipse to make EVOLIZER aware of its data models, so that they can later be accessed by other EVOLIZER plug-ins. For the approach presented in this paper, we plug in a custom-tailored implementation of the FAMIX model [TDD00] to represent facts about source code. Eventually, we need a data importer to extract and store information into the data model that we have registered. In case of our example, this is a parser that extracts the relevant facts from source code under analysis. The FAMIX source code meta-model and the fact extractor that we use are covered in more detail in the next section.

Data models in EVOLIZER are implemented using Java classes annotated with object-relational mapping meta-data according to the Enterprise JavaBeans 3.0 (EJB3.0) specification.[3] Persistency is provided through Hibernate,[4] a well-known high performance object/relational persistence and query service. As denoted earlier, EVOLIZER maintains a list of all registered data models and provides means to other EVOLIZER plug-ins to access the evolutionary information of a system via a convenient API.

Evolizer FAMIX

The FAMIX meta-model provides a language-independent representation of object-oriented source code [TDD00] and we use it in EVOLIZER whenever an analysis requires static source code information. An overview of the core model is given in Figure 5.2. It specifies the entities that can be extracted directly from source code.

The model defines important object-oriented relations as classes. For example, *Invocation* is explicitly modeled as a class instead of using a self-aggregation for *Method* (which would be implemented in Java as a collection of *Method*s as an Attribute of the class *Method*). This yields several benefits for us; the most important one is that we can map *Invocations* directly to the EVOLIZER RHDB and query them explicitly later. For example, we can retrieve all *Invocations* that fulfill certain properties, such as that they point to a particular method we are interested in, using HQL – the Hibernate Query Language:

```
from Invocation as invocation
    join invocation.callee as callee
where
    callee.name='addChart';
```

The results of such a query can then be used to, *e.g.*, provide a dependency analysis or visualization. EVOLIZER therefore already provides basic SQL-like query access, with features comparable to existing query languages for software analysis, to those that are familiar with both, HQL and the data available in the RHDB. On the other hand, the deficiencies of existing query languages also apply here; for developers that have no deeper knowledge of HQL and the underlying data structures, the information is hardly accessible through the data layer. In Section 5.3.3, we therefore describe how we add another layer to EVOLIZER to enable natural language interfaces.

We rely on a custom implementation of the FAMIX model, realized according to the procedure that we have outlined in Section 5.3.2. To populate an instance of the model with concrete data from source code under analysis, we use ZBINDER by Pinzger *et al.* [PGG07]. ZBINDER builds upon the Java Development Tools (JDT)[5] of Eclipse. The JDT parser alone fails in resolving cross references, such as method calls and attribute accesses of statements that contain a compile error – which is often the case when the code is incomplete or referenced libraries are missing. ZBINDER overcomes this limitation in most instances by gathering additional information stored in the abstract syntax tree and using sophisticated heuristics to reconstruct unresolved method calls.

[3] http://java.sun.com/products/ejb/
[4] http://www.hibernate.org/
[5] http://www.eclipse.org/jdt/

Static source code information for a software system can either be extracted from past releases that are stored within the EVOLIZER RHDB, or directly from a project that is currently under development in the workspace of Eclipse. ZBINDER can even be registered as a *builder* for a project, so that it gets notified every time a change is made to the source code.

The data layer of EVOLIZER provides a strong foundation for most of the classical software evolution and engineering analyses, especially when they depend on database performance, *e.g.*, interactive software visualizations. Knowledge processing tasks and tool integration, on the other hand, would benefit from formally defined data semantics that the data layer alone can not provide.

In the next section, we demonstrate by example how we overcome this limitation by adding an ontology layer to our platform.

5.3.3 Evolizer Ontology Layer

The EVOLIZER Java implementation of the FAMIX meta-model does not explicitly describe the formal semantics that is needed for automatic knowledge processing tasks such as query answering. For example, we can not define that there is an inverse relation between the property *declares Method* of a *Class* and the property *is declared in Class* of *Method*. If we are able to explicitly state the formal semantics then, every time we make a statement, such as *A declares B*, a semantic reasoner would be able to automatically infer that also *B is declared in A*. The Web Ontology Language OWL allows us to use description logic expressions to describe such relationships and to reason about them.

To take advantage of Semantic Web technologies, we added an additional layer on top of the EVOLIZER data layer by defining an OWL ontology that represents the FAMIX meta-model in terms of OWL classes, relationships and properties. This *source code ontology* is a subset of our software evolution ontology called `SEON`. Instance data is represented by RDF graphs. This way we get a knowledge base whose formal semantics enables automated processing. An overview of the ontology is shown in Table 5.1. The full ontology covers many more concepts, such as *interfaces*, *local variables*, *type casts* and usages of the *instanceof*-operator, as well as *exceptions*, but for the sake of this paper, we only focus on key concepts.

To populate the knowledge base, we need to map our Java implementation of the FAMIX meta-model to the OWL ontology. This mapping is done via a custom Java annotation `@rdf`. We add an annotation with the URI of the according OWL class to

Class: Class	**Class: Method**
→ hasMethod Method	→ accessesAttribute Attribute
→ hasAttribute Attribute	→ hasParameter Parameter
→ isReturnTypeOf Method	→ invokesMethod Method
→ isSubclassOf Class	→ hasReturnType Class
→ isSuperclassOf Class	→ isInvokedByMethod Method
→ hasName String	→ isMethodOf Class
	→ hasName String
Class: Attribute	
→ isAttributeOf Class	**Class: Parameter**
→ isAccessedByMethod Method	→ isParameterOf Method
→ hasName String	→ hasName String

Table 5.1: Simplified Version of the Evolizer Ontology for Source Code Analysis.

the signature of each Java class that has a counterpart in the ontology. Similarly, we annotate each Java method that should be mapped to a corresponding OWL relation or property name. We use Java reflection to automatically generate RDF statements from Java instances. This approach is similar to – and partially inspired by – the *so(m)mer*-project,[6] an Object-to-RDF mapping layer that uses annotations in the spirit of Hibernate. In Listing 5.1 we show an example of an annotated Java class. We have omitted the EJB3.0 annotations for persistency that are used by the data layer of EVOLIZER.

```
@rdf("http://evolizer.org/ont/java#Class")
public class FAMIXClass {
  @rdf("http://evolizer.org/ont/java#hasName")
  public String getName() {
    // ...
  }
  @rdf("http://evolizer.org/ont/java#hasMethod")
  public Set<FAMIXMethod> getMethods() {
    // ...
  }
  /* attributes, setter methods, and
     additional behaviour */
}
```

Listing 5.1: Java class annotated with @rdf.

[6] https://sommer.dev.java.net/

For the following discussion, we introduce the namespace prefix `evo:` as shortcut for `http://evolizer.org/ont/java\#`. In the example, the Java class `FAMIXClass` is annotated with the URI `evo:Class` and therefore, for every instance of the `FAMIXClass`, an instance of the OWL class `evo:Class` is generated in the RDF graph. This is done by creating a resource in the graph and adding a `rdf:type` property with `evo:Class` as value. The URI that represents the resource is generated using the unique database id maintained by the data layer of EVOLIZER.

In addition, for each annotated method in the Java class, a property is added to the resource. The return value of the annotated method is used as value of the property in the graph. Since the `getName()`-method has the return type `String`, the value is added as a literal. For the `getMethods()`-method the return type is a set of instances of `FAMIXMethod`. Since the return type is a `Set`, we add a property for each element in the set. The elements in the set are instances of the `FAMIXMethod` class (which is also annotated with `@rdf` annotations). Therefore, we trigger the generation of the RDF statements for the `FAMIXMethod` class as well, repeating the process described above.

An excerpt of the generated RDF graph is shown in Figure 5.3. Ellipses represent resources in the graph. The labels in the ellipses are the URIs that uniquely identify the resources. The labeled arcs represent the properties that relate the subject and the object of a RDF statement. Finally, literals are depicted as rectangles. In addition to the example that we have outlined above, we list also additional subgraphs that belong to the *version control* and *issue tracking* ontologies in Figure 5.3. This gives an impression on how we integrate different kinds of data sources, although we focus solely on knowledge covered by our code ontology in this paper.

In our Java-to-OWL mapping, we have to take a special case into account. Not all Java classes have counterparts among OWL classes. Java classes that explicitly model relationships are usually modeled as properties or relations in the ontology. In the RDF graph, they are represented as predicates. For example, the *Invocation*-class in the FAMIX model is modeled as *invokesMethod*-property in our ontology. We overcome this clash of paradigms by making it possible to mark a Java class explicitly to model a property by setting the flag `isPredicate` to `true`. In addition, we introduce two additional Java annotations `@subject` and `@object` to specify that, for example, in case of `Invocation`, the method `getCaller()` returns the subject and `getCallee()` the object of the RDF statement. The value of the `@rdf`-annotation is then considered to be the URI of an OWL-property, rather than of an OWL-class. This renders our mapping approach much more flexible, especially when the underlying Java class models were

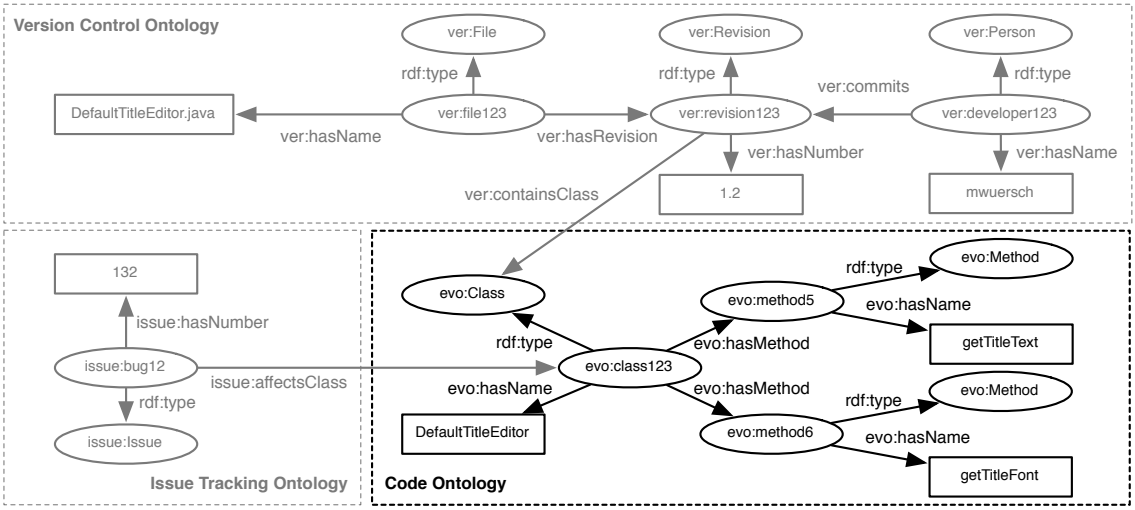

Figure 5.3: Excerpt of a generated RDF graph.

influenced by a relational view. An example of a class that models a property is given in Listing 5.2.

```
@rdf(
   isPredicate=true,
   value="http://evolizer.org/ont/invokesMethod"
)
public class Invocation {
    @subject
    public FAMIXMethod getCaller() {
       // ...
    }
    @object
    public FAMIXMethod getCallee() {
       // ...
    }
    /* attributes, setter methods, and
       additional behaviour */
}
```

Listing 5.2: Java class that models a property.

5.3.4 Evolizer Query Layer: Natural Language Querying with Ginseng

So far, we have discussed how we import static source code information (and facts about the evolution of a system in general) into our EVOLIZER RHDB. We also showed how we use ontology information to augment this data. We explained that we rely on the established industry standard OWL/RDF which enables us to leverage the potential of EVOLIZER with a plentitude of existing tools and frameworks for knowledge processing from academia, as well as from industry.

One such tool is Ginseng, a **g**uided **i**nput **n**atural language **s**earch **eng**ine, that was presented by Bernstein *et al.* in [BKKK06]. Ginseng benefits from the fact that ontologies are described in terms of triples of *subject*, *predicate*, and *object*. This structure strongly resembles the way how humans talk about things. It can be exploited to use quasi-natural language queries for accessing any OWL/RDF knowledge base. Ginseng is lightweight compared to classical full natural language interfaces since it uses no predefined vocabulary and queries are not interpreted logically or syntactically. Instead, the vocabulary is derived from the OWL knowledge base itself, *i.e.*, from labels of the instance data and from the identifiers that were used to define the ontologies. Optionally it is possible to add synonyms by annotating the ontology with Ginseng tags.

Ginseng uses a multi-level grammar consisting of a static part that defines basic sentence structures and phrases for English questions, and a dynamic part that is generated when an ontology is loaded. The static part additionally contains information on how to translate query sentences from quasi natural language to SPARQL. To generate the dynamic part of the grammar, the ontology is loaded into a Jena inferencing model and for each OWL class, individual (instance), object property, and data type property, a grammar rule is generated.

The full grammar is then used by Ginseng to guide its users by offering an auto-completion feature, *i.e.*, it presents a popup box with suggestions on how to complete the word that the developer is currently typing into the free-form input field. This limits the questions a developer can ask to a certain extent but prevents her from entering queries that are grammatically incorrect. Once the complete query is entered and concluded by a question mark, it is translated by Ginseng into SPARQL statements and executed against the knowledge base maintained by Jena. The results of the query are then presented to the developer.

Consider again the example graph given in Figure 5.3. If we want to query for all the

Methods that are declared in the class `DefaultTitleEditor`, we could ask:

What are the methods of DefaultTitleEditor?

That question does not need to be reformulated to a more formal query – it is accepted by Ginseng as it is listed above and developers additionally receive guidance in query composition when they start to type. By *guidance*, we mean that, in case of our example, as soon as the letter *'W'* is typed, all the words starting with that letter are listed in a drop-down menu (see Section 5.4.1 for a full, illustrated example). Ginseng continues to do so until a complete and valid sentence (in terms of that it satisfies the grammar rules) was entered and will then automatically continue with translating the question into the following SPARQL query:

```
SELECT ?methods
WHERE {
   ?methods <rdf:type>      <evo:Method> .
   ?class   <evo:hasMethod> ?methods .
   ?class   <rdf:type>      <evo:Class> .
   ?class   <evo:hasName>   "DefaultTitleEditor" .
}
```

When the query above is executed, the graph pattern consisting of the four triple patterns in the `WHERE`-clause is matched against the triples in the RDF graph, and returns the bindings for the variables in the `SELECT`-clause. In SPARQL, variables are indicated by the prefix "?". The predicate and the object of the first triple are fixed values, so the pattern is going to match only triples with those values. Within a graph pattern a variable must have the same value no matter where it is used, so the query above only returns a binding for `?method` if the variables called `?method` in the first and second triple have the same value; as well as the variables called `?class` in the second, third, and fourth triple.

For more details on Ginseng and its underlying technology, we refer to [BKKK06].

5.3.5 Wrapping up: The Integration of the three Layers of Evolizer

When we want to query for facts about source code, we tell the data layer of EVOLIZER to parse the currently selected project in the Eclipse workspace (alternatively it is possible to process the code of a past release stored within the RHDB). The most convenient way to trigger this process is to add a EVOLIZER *Nature* to an Eclipse project. Along with the *nature*, a builder is then assigned to the project that automatically re-builds the

FAMIX-model every time a change to the source code is made. Re-building the FAMIX model means that the source code is parsed by `ZBinder` which creates instances of `FamixClass`, `FamixMethod`, `Invocations`, and so on, according to the facts that it finds within the source code. Then it stores these instances into the RHDB.

The data is then passed to the ontology layer which translates it according to the `@rdf`-annotations and the OWL description of the *Evolizer Ontology for Source Code Analysis* into a Jena Ontology model.

This model is then analyzed by Ginseng and, subsequently, available to the developer for querying in natural language. The results are presented to the developer in an Eclipse view, similar to that provided by Eclipse itself for displaying Java search results. Since we also keep track of source code locations in our model, the developer can easily navigate from the results view directly to the corresponding source code.

Next, we provide a case study to demonstrate how our prototype implementation of the framework described above can be used by a developer to answer common questions during daily program comprehension tasks.

5.4 Case Study

In our case study, we demonstrate by the example of the open-source library JFreeChart[7] that our framework can be used to answer the most common program comprehension questions that arise during software evolution tasks [dAM08]. We do not focus on evaluating the quality of the query results – as we have explained throughout the preceding sections, the data importers are not the key contribution of this paper and can be exchanged easily thanks to EVOLIZER's plug-in architecture. Instead, we show that, compared to existing tools, developers are given more flexibility when composing queries with our approach: they can formulate queries conveniently using different variations of natural language sentences.

In [KB07], Kaufmann *et al.* presented a usability study with 48 users, evenly distributed over a wide range of backgrounds and professions, including software development. The study incorporated four query interfaces (including Ginseng) featuring four different query languages that demonstrated the usefulness of natural language interfaces for casual end-users. Their experiment was based on geographical data encoded in an OWL knowledge base. Kaufmann *et al.* found that:

[7] http://www.jfree.org/jfreechart/

> *"(1) With full-sentence questions, users can communicate their information need in a familiar and natural way without having to think of appropriate keywords in order to find what they are looking for. (2) People can express more semantics when they use full sentences and not just keywords." [KB07]*

Although we did not yet conduct an extensive user study in the software engineering domain, we claim that the results of this study are, to some extent, applicable to our setting. It is reasonable to assume that the domain of the knowledge that we query can be neglected, compared to the influence of the professional background of the users and, as a consequence, their familiarity with more formal languages. The study of Kaufmann *et al.*, however, showed that the findings above apply to both categories of users likewise – to those without any prior knowledge of query languages, as well as to those with a background in software engineering and familiar with at least SQL.

5.4.1 Using Evolizer to answer common Program Comprehension Questions

In [dAM08], De Alwis and Murphy have listed 36 common program comprehension questions that their tool *Ferret* implements. The questions fall into five categories: *inter-class*, *intra-class*, *inheritance*, *declarations*, and *evolution*. The questions are further assigned to one or several contexts, or what they call contributing *spheres*: The *static*-sphere relies on static program analysis, the *dynamic*-sphere uses profiling information, the *evolution*-sphere relies on software repository mining, and the *plug-in*-sphere contains declarative information specified in Eclipse plug-in manifests.

EVOLIZER supports all of their static queries *out of the box*, without having them predefined or hard-coded explicitly. Conceptually, we can also answer all the questions from the *evolution*-sphere.

In the following we have selected, for each of the first four categories, those questions that proved to be most useful to developers in the field-study conducted by De Alwis and Murphy. We use them to exemplify how EVOLIZER can be used to support program comprehension. As a case study, we use Release 1.0.12 of JFreeChart, an open-source chart library written in Java with a size of more than 250 kLOC.

Questions concerning Inter-Class Dependencies

De Alwis and Murphy identified the question *"What calls this method?"* to be the most commonly asked one when a developer is trying to understand a system. The question falls into the category of inter-class dependencies and can be easily answered with EVOLIZER, as well as by many existing tools – including Eclipse itself. We have randomly chosen the class `ChartDeleter` from JFreeChart. The class declares four methods, among them `addChart(String)`. To find its callers, we can enter exactly:

What methods call addChart?

into the input field of Ginseng and execute the query. Figure 5.4 illustrates how Ginseng provides guidance for the developer to compose a query: When she starts to type *"W"*, a list of possible question words pops up (Step 1). After selecting the word *"What"*, she types *"a"* and receives several suggestions starting with that letter, such as *"accesses"*, *"are"*, *"arguments"*, and so on. Going on like that, the developer is able to compose the complete query (Steps 3 to 6) and, as soon a valid query was entered, she can execute it by concluding the sentence with a question mark (Step 7). This kind of guidance is especially valuable for novice programmers, who are not already familiar with the underlying knowledge base.

In case of JFreeChart, a single match is presented after the execution of the query: `registerChartForDeletion(File, HttpSession)` of the class `ServletUtilities` (Step R). This corresponds to the result of invoking the *"Find references in project"*-functionality of Eclipse.

Variations of the initial question are also possible:

What are the callers of addChart?

is accepted by Ginseng just as well as the imperative form:

Give me the methods invoking addChart!

We want to remark that Ginseng automatically generates these variations solely based on its grammar rules, synonyms encoded in the ontology, and instance data provided by EVOLIZER. There is no need to explicitly define in advance the questions that are possible – developers can ask them based on the facts extracted from source code.

Another program comprehension question that was identified to be asked often by developers is *"What fields are declared as being of this type?"* The structure of the original question is too complex for the simple grammar rules of Ginseng but can be reformulated

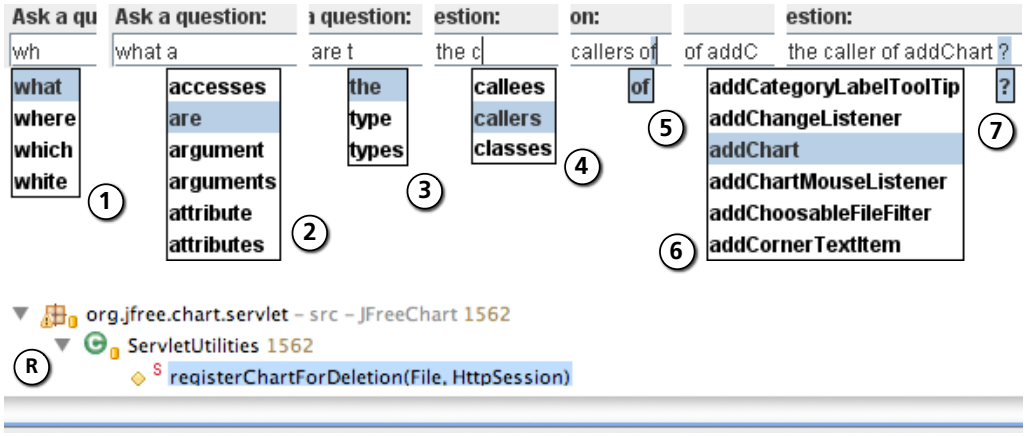

Figure 5.4: Entering a query and retrieving the result.

to: *"What fields have the type <type>?"* For example, we can search for attributes of type `JTextField` to identify classes of JFreeChart that contribute to the user interface in general and, in particular, process user input. Entering the question:

What attributes have the type JTextField?

returns seven results; three in `DefaultAxisEditor`, two each in `DefaultNumberAxisEditor` and `DefaultTitleEditor`.

This time significantly more user interaction is necessary to come up with the same results using the *Java Search* of Eclipse. Besides entering the search string, we have to choose *Search for type* and configure the *"Limit To"* option to only match for *field types*. Especially newcomers to Eclipse are often not aware of these features.

Questions concerning Intra-Class Dependencies

During maintenance tasks, developers often face god-classes several thousand lines of code in size. Changing, *e.g.*, the type of one of their attributes is a tedious task, involving careful code investigation and a lot of scrolling to answer questions, such as *"What methods access this field?"* Often, the task is further complicated when information hiding principles were violated. Using our framework, we are able to significantly narrow down the amount of code that has to be inspected. Coming back to one of the examples in the last section, we can ask what methods access the field `labelFontField` in `DefaultAxisEditor`. Again, the user is guided during query composition. When she starts typing *"What method accesses..."*, Ginseng automatically suggests *attribute* (as well as field, as a synonym) and

method as the next possible words. By choosing *attribute*, the set of suggested candidates for the concluding word is reduced to the names of particular fields, *i.e.*, names of methods are faded out. The full query is:

What method accesses the attribute labelFontField?

Executing the query yields two results: the constructor of `DefaultAxisEditor` and the method `attemptLabelFontSelection()`. Manual inspection confirms these results to be correct.

Questions concerning Inheritance

Generalization and specialization are among the most powerful features in Object-Orientation. The other side of the coin is that inheritance increases the gap between the static program and the dynamic process and therefore complicates program understanding in some cases, especially if used too extensive (*e.g.*, deep inheritance hierarchies over more than seven levels), or incorrectly (*e.g.*, to simply reuse code, rather than backed by the idea of specialization). Supporting the developer with tools to understand inheritance hierarchies more easily is therefore desirable. Browsing through code and navigating upwards the inheritance hierarchy is already well-supported by modern IDEs. Navigating downwards, on the other side, usually involves tedious searching. However, literally speaking, it could be as easy as asking *"What are the subclasses of this class?"*, if our approach were used.

The class `DefaultAxisEditor` in JFreeChart is implemented as a subclass of `JPanel`. Querying the static source code information in FAMIX, we can quickly locate similar classes, *i.e.*, classes that extend `JPanel`: `DefaultTitleEditor`, `DefaultAxisEditor`, and `FontChooserPanel`, among others. If the underlying ontology provides meaningful synonyms, each of the following queries would return the information that we are interested in:

- *What are the classes that extend JPanel?*
- *What are the subclasses of JPanel?*
- *What classes inherit from JPanel?*

Questions concerning Declarations

We have chosen *"What are all the fields declared by this type?"* among the 36 conceptional queries that De Alwis and Murphy have listed in their paper to conclude our case study

on how developers can benefit from our framework. The question has been identified by De Alwis and Murphy to be the most commonly asked one concerning declarations. In the last example, we have identified a few classes that are subclasses of `JPanel`. If we want to confirm the initial impression that `DefaultTitleEditor` and `DefaultAxisEditor` implement similar concepts according to their naming scheme, we can do that in terms of comparing their states, *i.e.*, fields. Investigating the answers to the queries:

What attributes are defined in DefaultAxisEditor?

and:

What are the attributes of DefaultTitleEditor?

confirms that they, in fact, have similar states: For example, both of them seem to have associated a font (`labelFont` and `titleFont`, respectively), a checkbox (`showTickLabelsCheckBox` and `showTitleCheckBox`, respectively), and so on. The next steps would probably be to investigate further the types of the fields and the behavior that operates on them or to have a look at the documentation of the two classes, *e.g.*, by entering the following query:

Give me the Javadoc of DefaultAxisEditor!

From here, gaining a deeper understanding of JFreeChart is just a matter of asking the right questions.

5.4.2 Discussion and Limitations

We conducted a validation of our approach by addressing the questions that De Alwis and Murphy listed in their paper about Ferret. The two approaches share many similarities in terms of their goals. Furthermore, those questions were identified in two empirical studies [SMV08,SMV06] to be the most frequently asked questions by programmers during software evolution tasks and therefore provide a suitable benchmark for our framework.

Our approach, in contrast to Ferret, can draw from the full power of the semantic web technologies and is therefore not limited to a set of predefined queries, with the need to have any additional ones implemented by some provider, such as a tools-support group. Instead, the querying capabilities of our framework are much more flexible and only limited by a subset of the English grammar and by the knowledge base that is available. Therefore the developer queries that we have chosen in our case study are only a subset of the ones

that can be formulated and answered *out of the box*. Many more are possible and they can be formulated in different variations, *e.g.*, as a questions or using the imperative form.

IDE vendors need to provide an interface to the information offered by tools comparable to Ferret (often menu-items in deeply nested context-menus). This becomes more and more of a problem as the information need of developers may become more diverse with the increase in complexity of modern software systems. Our framework, in contrast, provides a single access point for most of the information needs: using natural language, a developer can just ask what is on her mind, without having to worry where the desired functionality is hidden in the deeply nested menus of her favorite IDE.

Existing query languages for software evolution artifacts potentially also provide such an access point and give the developer the freedom to formulate queries without being bound to a set of predefined ones. On the other hand, they usually rely on custom-tailored, verbose languages. Therefore, they are hardly used in practice. A simple question, such as *"What methods call addChart?"*, which our tool answers right away, has to be reformulated by a developer into a SQL-like statement in order to be answerable with a tool, such as Semmle.[8] A Semmle query would look like this:

```
from Method caller, Method callee
where caller.calls(callee)
and   caller.fromSource()
and   callee.fromSource()
and   caller != callee
and   callee.getName().matches("addChart
select caller.getName()
```

Moreover, extending existing query languages with new vocabulary involves manual editing of the language definition files, whereas in our framework, additional vocabulary is available as soon as new data is loaded into EVOLIZER. This is possible because Ginseng generates dynamic grammar rules from the loaded OWL ontologies, but it also implies that we have to rely on meaningful identifiers in the ontologies that we query. If this is not the case, we also have to fall back to manual definitions of synonyms. This is straight-forward and can be done either in advance by a tool-vendor or later by the end-users, *i.e.*, developers – even if they are unfamiliar with the Semantic Web – in case that they are more comfortable with another vocabulary than the one that is already provided by the ontology.

Query languages are less ambiguous than natural language in general and therefore better in expressing, *e.g.*, complex restrictions and in formulating composed queries. However, supported by the empirical studies mentioned above, we claim that the most common program comprehension questions have a simple structure. In rare cases where

[8] http://www.semmle.com/

more expressive power is needed, one can always fall back to SPARQL or to the SQL-like Hibernate Query Language (HQL).

The performance of our prototype on a common laptop computer is acceptable for a project of the size of JFreeChart (∼250kLOC, the response time for the queries presented in our case study was usually around a couple of seconds).

5.5 Related Work

Our work is highly related to the approach of De Alwis and Murphy presented in [dAM08]. Just like them, we offer a framework to support the composition and integration of different sources of data about software artifacts in a single queryable knowledge base. In contrast to our approach, they define their own *sphere model*, whereas we rely on standardized technologies that are already established in the research community, as well as in industry. Moreover, while Ferret restricts developers to a set of predefined, hard-coded questions, we give them the freedom to formulate their own questions by exploiting existing natural language query tools.

Natural Language in Program Comprehension

LaSSIE, presented by Devanbu *et al.* in [DBS91], integrated multiple views on a software system at AT&T in a frame-based knowledge base and also provided semantic retrieval through a natural language interface. LaSSIE and our framework share many commonalities, especially since the Semantic Web emerged from frame-based knowledge representation techniques. Hill *et al.* presented an algorithm to extract noun, verb, and prepositional phrases from method and field signatures in source code to enable *contextual searching* [HPVS09]. The queries they support are closer to keyword search on identifiers found in source code than to full natural language questions and they do not cover structural information, such as caller-callee or inheritance relationships among source code entities.

Query Languages for Software Artifacts

Many approaches have been proposed that use specific languages to query software artifacts. They are either based on standard database languages, such as SQL or Datalog (*e.g.*, CodeQuest [HVdM06] and Semmle), customized Prolog implementations (*e.g.*,

JQuery [JV03], ASTLog [Cre97], GraphLog [CMR92]), or a custom language (*e.g.*, SCA [SA96]). All of them aim to help developers to effectively explore and better understand code, uncovering information that would be impossible or extremely hard to find with standard tools. However, most of them require the user to master syntax and vocabulary of a specific query language limited to that single purpose. Our approach guides developers in vocabulary, as well as in syntax, to construct well-formed and coherent questions about static source code information. Nevertheless, we consider most of these query languages complementary to our approach, as they are more expressive in terms of that it is possible to compose more complex queries than with the subset of English grammar rules that we rely on. In general, as argued in Section 5.4, the most common questions that arise during software evolution tasks are of simple structure and are therefore predestinated to be answered with natural language using EVOLIZER.

Semantic Web in Software Engineering

Our framework relies heavily on Semantic Web technologies. Besides the Web, these technologies have proven to be useful in many domains, for example to enable the interoperability of software systems, and when technologies are needed to express knowledge with formal semantics to enable machine processing. Software Engineering is one of these domains. An overview of applications of ontologies in software engineering has been given in [HS06, GL02, UJ96]. All of these publications promote the theoretical benefits offered by different characteristics of ontologies, such as explicit semantics and taxonomy, lack of polysemy, ease of communication and automatic data exchange between distinct tools, and computational inference. On the other hand, only few approaches put those ideas into real practice. Hyland-Wood *et al.* [HWCK06] propose an OWL ontology of software engineering concepts (SEC), including classes, tests, metrics, and requirements. Bertoa *et al.* [BVG06] follow a similar approach but focus more on software measurement. Happel *et al.* [HKST06] propose various ontologies to foster software reuse. In their KOntoR approach, they provide therefore background knowledge about software artifacts, such as the programming language and licensing models. Kiefer *et al.* developed EvoOnt, a software repository data exchange format based on OWL [KBT07]. Their software ontology model (SOM) was influenced by FAMIX. Their version ontology model (VOM) and their bug ontology model (BOM) are based on EVOLIZER's data models for CVS and bug tracking information, respectively. The authors use iSPARQL, their extension to the RDF query language SPARQL, for effectively querying their ontologies to detect code smells.

Witte *et al.* [WZR07] use text mining and static code analysis to map documentation to source code for software maintenance purposes. These mappings are represented in RDF. The MOST project [mos08] aims to facilitate software engineering by leveraging ontology and reasoning technologies. It integrates ontologies into model-driven software development (MDSD), resulting in ontology driven software development (ODSD). All of the approaches mentioned above acknowledge the potential of ontologies and the Semantic Web applied to software engineering. They often define custom ontologies that can be integrated in the ontology layer of EVOLIZER.

5.6 Conclusions

As software systems get more complex, efficient tools to support software engineers during their development and maintenance tasks are becoming more important. Modern IDEs already made a great leap forward in providing a variety of features to, for example, facilitate program comprehension. The complexity of the user interface is putting a significant cognitive burden on a developer. Often it is easier to solve a task manually than to master a tool. Although experienced developers usually know exactly what information they are looking for, they often do not know how to get it. They simply do not know how to turn conceptual queries into something their IDE understands.

In this paper, we presented a framework that overcomes this gap and showed an application of Semantic Web technologies that goes beyond merely data exchange for the sake of tool interoperability. We combined industrial-strength technologies with ideas and tools from the Semantic Web to enable developers to query software engineering artifacts in a way that is familiar to them: using (quasi) natural language strongly resembling plain English. For that, we use the Web Ontology Language OWL to describe static source code information extracted by our EVOLIZER. The resulting ontology then serves as input for the guided-input natural language interface Ginseng. We demonstrated in a case study that our approach makes it possible to answer the most common program comprehension questions identified in the literature.

We do not restrict developers to a set of predefined questions but advance the state-of-the-art in that our approach is only dependent on what data is available as input. With our framework, it is straight-forward to integrate almost any kind of evolutionary information, for example, from version control or issue tracking systems – solely by exploiting existing and well-established standards for resource description. We encourage other researchers to

download and try out our EVOLIZER toolset or to incorporate our SEON ontology into their own tools.

5.7 Acknowledgements

This work was supported by the Hasler Foundation, Switzerland as part of the *ProMed-Services* project and the Swiss National Science Foundation as part of the *Enterprise Computing* and *diCoSA* projects. We also thank Esther Kaufmann for providing us Ginseng.

6

Software Evolution Analysis Composition in *SOFAS*

A Framework for Semi-Automated Software Evolution Analysis Composition
Giacomo Ghezzi and Harald C. Gall

Submitted to the Automated Software Engineering journal

Software evolution data stored in repositories such as version control, bug and issue tracking, or mailing lists is crucial to better understand a software system and assess its quality. A myriad of analyses exploiting such data have been proposed throughout the years. However, easy and straight forward synergies between these analyses rarely exist. To tackle this problem we have investigated the concept of *Software Analysis as a Service* and devised *SOFAS*, a distributed and collaborative software evolution analysis platform. Software analyses are offered as services that can be accessed, composed into workflows, and executed over the Internet. This paper presents our framework for composing these analyses into workflows, consisting of a custom-made modeling language and a composition infrastructure for the service offerings. The framework exploits the RESTful nature of our analysis service architecture and comes with a service composer to enable semi-automated service compositions by a user. We validate our framework by showcasing two different approaches built on top of it that support different stakeholders in gaining a deeper insight into a project history and evolution. As a result, our framework has shown its

applicability to deliver diverse, complex analyses across system and tool boundaries.

6.1 Introduction

Until recently, historical data stored into repositories such as version control, bug and issue tracking, or mailing lists had been mostly neglected or considered a necessary byproduct of software development. However, studies have highlighted the value of collecting and analyzing these diverse sources of data [MV00, uM03, ZWDZ04]. This sparked what can be considered a "gold rush" to mine all sorts of useful information. A growing number of analysis techniques, such as static and dynamic code analyses, code clone detection, co-change analysis, bug prediction or detection of bug fixing patterns, have been devised. Yet, despite this richness, the issue of easy and straightforward integration and sharing of data produced by different analyses has been left almost entirely unaddressed.

The use and combination of different software analyses is still a challenging problem when trying to gain a deeper insight into the history of a software system. Moreover, the replication of software evolution empirical studies is negatively affected. In fact, as shown by Robles [Rob10], both the analyses and their results, even when available, are rarely usable for replication in an effective way. Because of this, even though software evolution research has a strong foundation on empirical studies, a systematic framework enabling replicability is still missing. We claim that this status quo severely hampers the progress of software evolution research and its soundness.

To tackle this problem, we introduced the basic concept of *Software Analysis as a Service* [GG08]. Based on that, we devised a RESTful analysis architecture called *SOFAS* (SOFtware Analysis Services) [GG11]. It provides the foundations for distributed analysis services, which enable a lightweight interoperability of analyses across platforms and geographical or organizational boundaries. *SOFAS* consists of three main constituents: Software Analysis Web Services (*SA-WS*), Software Analysis Ontologies (*SA-Ontos*) and a Software Analysis Broker (*SA-B*). *SA-WS* offer different software evolution analyses as standard RESTful web service interfaces. They adhere to specific meta-models and *SA-Ontos* that define and represent the data they consume and produce. The *SA-B* acts as the services manager and the interface between the services and the users. It contains a *Services Catalog* of all the registered analysis services with respect to a specific software analysis taxonomy.

In our previous papers [GG08, GG11] we sketched the basic idea of the approach and its architectural design. This paper presents a framework for semi-automated software analysis composition that we integrated into *SOFAS*. We explain how this composition works and describe *SCoLa*, a new language we devised to define the composition of

analyses and model workflows. We introduce two concrete applications of these workflows built on top of this framework, used to investigate different aspects of the evolution of a software system. With these two applications we demonstrate the versatility of our approach in helping software evolution researchers to systematically gain a deeper and wider insight in the history and quality of software systems.

To shed light on the analysis context that we address, consider, for example, the task of getting an overview on the evolution of a specific project, e.g. Apache Tomcat[1], using well-known indicators such as source code metrics, code clones and change coupling among the project files. To get that data, we would normally need to (1) download all the source code; (2) find and set up an appropriate metrics calculator (e.g. Metrics[2] or Imagix4D[3]) and a code clone detector (e.g. JCCD[4] or CCFinder[5]) and feed them the code; (3) analyze the project's SVN repository to calculate the change coupling of all its files; (4) depending on the tools used, manually interpret their results and aggregate them accordingly. This would involve dealing with different explicit and implicit meta-models and formats used to represent the data produced by the different tools. Moreover, if this process were to be repeated using any different tool, the last step would have to be redone from scratch. With our approach we can assemble a workflow that takes the project source, the version control repository URL, the clone detection strategy settings (*e.g.*, the threshold number of tokens after which a piece of code is considered a clone) and runs the exact same process automatically. Due to the use of ontologies, the resulting data are semantically and syntactically defined, regardless of the actual metrics, code clones or change coupling analyses used (as long as they belong to the same categories). Furthermore, more analyses making use of the produced data can be added. For example, one that given the extracted source code metrics, finds all the relevant code smells, as done by Lanza *et al.* [LM05]. At last, the results can be further analyzed and refined using SPARQL (or other filters and aggregators). For example, one can automatically assess if the amount of duplicated code or the value of some specific metric exceeds a certain threshold.

In Section 6.2 we give a brief overview of *SOFAS*. For a detailed description of the architectural aspects we refer to [GG11]. Section 6.3 introduces *SCoLa*, our custom composition language: its main components and how its workflows are created, checked for validity, and executed. Section 6.4 describes how *SCoLa* is integrated and used in *SOFAS*.

[1] http://tomcat.apache.org/

[2] http://metrics.sourceforge.net/

[3] http://www.imagix.com/products/source-code-analysis.html

[4] http://jccd.sourceforge.net

[5] www.ccfinder.net

In Section 6.5 we show two concrete applications of these workflows to better assess their potentiality and versatility. Section 6.6 gives an overview of the related work, in particular evolution analysis composition and RESTful services description and composition. We then conclude with a discussion on the strength and weaknesses of the approach and the possible future directions.

6.2 SOFAS

SOFAS is a RESTful architecture offering a simple yet effective way to provide software analyses. It is based on the principles of Representational State Transfer around resources on the web (as introduced by Fielding [Fie00]). Figure 6.1 gives an overview of the architecture, which is made up by three main constituents: Software Analysis Web Services (*SA-WS*), a Software Analysis Broker (*SA-B*), and Software Analysis Ontologies (*SA-Ontos*). *SA-WS* expose the functionality and data of software (evolution) analyses through a standard RESTful web service interface. The *SA-B* acts as the services manager and the interface between the services and the users. It contains a catalog of all the registered analysis services. Ontologies define and represent the data consumed and produced by the different services. In the following, we briefly describe each of these three components.

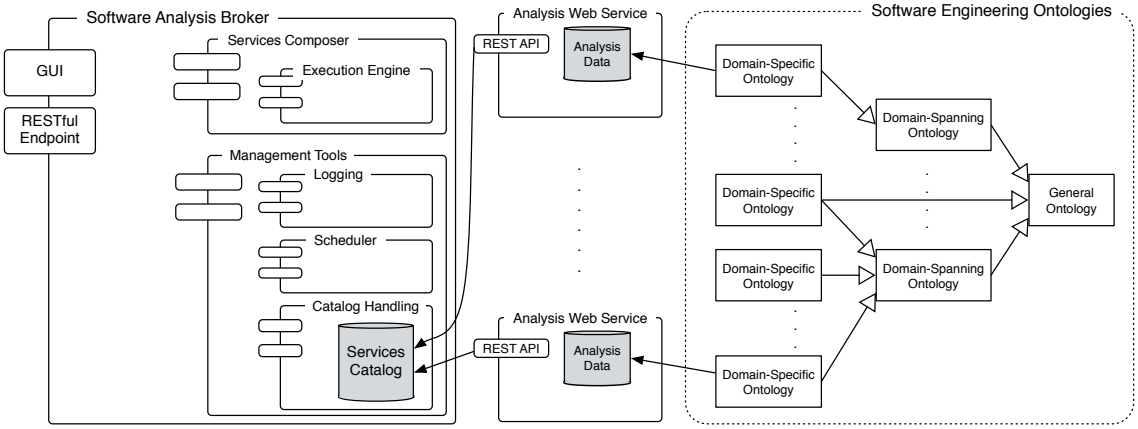

Figure 6.1: *SOFAS* overall architecture

6.2.1 Software Analysis Web Services

SOFAS' purpose is to provide software analyses and the data they produce in a simple, standardized way, freeing them from specific IDEs, platforms and languages. From a user's perspective, software analyses are inherently linear and uniform in the way they work. Given some information about a software project (be it the code, its source code repository, some data already calculated by an analysis, etc.) and possible analysis calibration settings, they extract and/or calculate their specific data. Once that is completed, the results can be fetched in different, specific formats and, when needed, they can also be updated or deleted. Given these premises, RESTful services perfectly fit our needs. The main requirements and characteristics of our services are indeed some of the main inherent principles of REST.

A RESTful web service provides a uniform interface to the clients, no matter what it actually does. It is a collection of resources all identified by URIs, which can be accessed and manipulated with HTTP methods (e.g., POST, GET, PUT or DELETE). Furthermore, every message exchanged is self-descriptive as it always contains the Internet media type of the content, which is enough to describe how to process it. In our case, the analyses services boil down to simply two resources: the service itself and the individual analyses.

These analyses can be classified into three categories: (1) data gatherers; (2) basic software evolution analyses; and (3) composite software evolution analysis.

Data gatherers

Data gatherers work on raw data to extract evolution information from different software repositories, such as version control, issue tracking, mailing lists, or plain source code, and import it into *SOFAS* for other analyses to use it. Gathering this data can be extremely time consuming, as project histories can consist of several years of active development (*e.g.*,, Firefox version control history consists of more than 95.000 commits spread over 10 years). However, this is a vital step for any analysis, as it provides the necessary software project data to work on. As of now, the following data gatherers are registered in *SOFAS*:

1. Version history importers for CVS, SVN, GIT and Mercurial. They extract the version control information comprising release, revision, and commit data from a given version control repository.

2. Issue tracking history importers for Bugzilla, Google Code, Trac, and SourceForge. They extract the issue tracking history from a given issue tracker instance.

3. GNU Mailman importer. It extracts communication data from a given GNU Mailman-based mailing list.

4. Meta-model extractors for Java and C#. They extract the static source code structure of a software project, based on the FAMIX meta-model [TDD00].

Basic software evolution analyses

Basic services exploit the data imported by one data gatherer to calculate all sorts of software evolution information: version history metrics, code metrics of specific releases/revisions, issue tracking metrics, etc. The analyses currently registered are:

1. Version history metrics calculator. It calculates several statistics from a given project version history.

2. Release meta-model extractor. It extracts the static source code structure (based on FAMIX) of one or more specific releases of software project (written in Java or C#), given its extracted version history.

3. Code Metrics calculators. They compute some of the most common software metrics (35 as of now) of a software system. The entire list of the metrics offered, along with a brief description of them, can be found on the web[6][7].

4. Change type distiller. Given a project version history, it extracts, for each revision, all the fine-grained source code changes of each source code file. These changes are then classified following the change types taxonomy proposed in [FWPG07].

5. Change coupling detector. It calculates the change couplings for all the files from a given version control history, as described by Gall *et al.* [GJK03].

6. Change coupling history calculator. It calculates the evolution of change couplings over the duration of a given version control history.

7. Code clones detector. It extracts the code clones from a specific version of a given version control history using JCCD.[8]

[6] http://habanero.ifi.uzh.ch/famixMetrics
[7] http://habanero.ifi.uzh.ch/javaFamixMetrics
[8] http://jccd.sourceforge.net

8. Code clones history calculator. It extracts the code clones from a given version control history, by regular intervals defined by the user.

9. Yesterday's Weather service. It calculates the Yesterday's Weather metric [GDL04] from a given version control history.

10. Code ownership detector. It detects, for each file, which developers "own" it. That is the developers who should know the most about that specific file, based on how much and when they changed it. This information is extracted from a given version control history.

11. Gini coefficient calculator. It calculates the distribution of changes between the developers in a given version control history using the Gini coefficient [Gin12], as proposed by Giger et al. [GPG11b].

12. Change metrics calculator. It calculates the file-level change metrics proposed by Moser et al. [MPS08] from a given version control history.

13. Change metrics-based defect predictor. It calculates the most defect prone files based on the change metrics calculated by the aforementioned change metrics analysis.

Composite software evolution analyses

Composite services aggregate data produced by other analyses to calculate more complex and domain spanning evolution information. These are some of the analyses currently registered in *SOFAS*:

1. Issue-revision linkers. Given the issue tracking and version histories of a specific software project, they reconstruct the links between issues and the revisions that fixed them. As of now three of them exist, using the heuristics proposed by Mockus *et al.* [MV00], Sliwerski *et al.* [SZZ05b], and Fischer *et al.* [FPG03b].

2. Code Disharmonies detector. It detects all the code disharmonies [LM05] in a software project using the code metrics extracted by the aforementioned metrics calculators.

3. Code-churn-based defect predictor. It predicts the most defect prone entities based on the combination of source code metrics calculated for specific snapshots of a given version control history. It is based on the algorithm proposed by D'Ambros et al. [DLR10].

4. Bug Cache defect predictor. Given the issue tracking and version histories of a specific software project and the links between them (detected by one of the aforementioned linkers), it predicts further faults, based on the algorithm developed by Kim et al. [KZJZ07].

5. Email-Source code linker. It links emails with source code given version history and mailing list information extracted by the associated data gatherers. It uses the algorithm proposed by Bacchelli et al. [BLR10].

6. Metrics-based defect predictor. It predicts the most defect prone entities based on the combination of source code metrics calculated by the aforementioned analyses.

6.2.2 Software Analysis Ontologies

To semantically describe the data produced by software analyses, we have developed our own family of ontologies, called *SEON* (Software Engineering ONtologies)[9]. An ontology is a formal description of the important concepts (classes of objects) identified in the domain of discourse and their relationship to one another [Gru93]. It provides a common vocabulary for a specific domain, which can be used to express the meta-data needed to capture the knowledge of the exchanged, shared, or reused data. Our ontologies, defined in OWL, are organized in a pyramidal structure. At the bottom of the pyramid sit ontologies describing system-specific or language-dependent concepts (*e.g.*, Java-specific language constructs, SVN-specific versioning concepts, Jira-specific issue tracking concepts, etc.). The second layer defines domain-spanning concepts that were abstracted from system or language specifics. This layer contains concepts and relationships for version control, issue tracking, or some object-oriented programming languages like Java and C#. The top layer is comprised of higher-level ontologies describing general concepts, the attributes to describe them, and the relations between the concepts. We refer to *SEON*'s web page for a complete description of these ontologies.

6.3 Software Analysis Composition

The use of different analyses by themselves can already uncover vital information about a software project, as already shown by works such as [BWKG05, NB05, ZWDZ04].

[9] www.se-on.org

However, it is the ability to combine them into workflows, and thereby building a much broader understanding of software and its evolution, that sets the use of services apart form the current state of the art. In our case, it allows us to concatenate data gathering services with the analyses exploiting the data they produce. To effectively compose and execute these workflows, a web service composition language was required. Several of such languages have been proposed such as BPMN [bpm11], WSCI [wsc02], WS-CDL [wsc05] or WS-BPEL [JE07b], with the latter emerging as the most successful and widespread.

All these languages were created with classic SOAP RPC-based services in mind. One of the main features of our solution is the use of RESTful services, which significantly differ from the former. This makes those languages hardly usable. Custom solutions, such as extending WS-BPEL to account for REST [Pau08], describing RESTful services with WSDL 2.0 [Man08] or creating new ad-hoc languages and tools [Pau09, ZD09] have been recently proposed. However, they have not really gained ground or have not been used outside theoretical case studies. This is also due to the fact that the majority of the existing RESTful services still rely on human-oriented documentation. Besides, there has not been a concrete and widespread need to compose RESTful services yet.

Using a full-fledged approach based on WS-BPEL or on a similar solution would, in our opinion, add unnecessary complexity to something that has simplicity, uniformity, and ease of use as its main features. Furthermore, *SOFAS*' services, by being RESTful do not only have the same interface, but they also exhibit the same behavior. Analyses can be started, managed and the outcome data be fetched always in the same manner. This allows us to make several additional assumptions and simplifications in modeling how they work and how they can be composed. In particular, a workflow always consists of starting one or more analyses (an HTTP post method on the service URL), waiting for them to finish (repeatedly calling an HTTP head method on the analysis URL) and, when done, passing the URI of the results directly to waiting analyses (along with analysis specific options) or querying the results to fetch some specific data to pass to the waiting analyses—and so on, until the workflow is completed.

As a consequence, a viable solution was to develop a custom service composition language, which we called *SCoLa* (*SOFAS* Composition Language) instead of using any of the existing standards. *SCoLa* is a simplified and modified version of WS-BPEL.

In the following, we will quickly go through all the fundamental components of it assuming that the reader already has some knowledge of its parent language. Please note that a detailed, formal description of the language is beyond the scope of this paper.

6.3.1 An Overview of SCoLa

SCoLa is intended for modeling an executable workflow of software analysis services, specifying the execution order between a number of constituent activities, the partners involved and the messages exchanged between these partners.

A workflow definition has two main sections. The *variables* section defines the data variables used in the workflow, providing their definitions in terms of XML Schema types (simple or complex). Variables allow workflows to maintain information between service calls and pass data produced by one service to another. The remainder of the description contains the actual workflow's behavior, which is, as in its parent language WS-BPEL, a kind of flow-chart. Each element in the process is called an *activity*. An activity is either *primitive* or *structured*. The primitive activity types are:

- **invoke** to invoke a service to start its specific analysis, given some input.
- **query** to query the results of an analysis with a SPARQL query passed as input and save the results into a variable.
- **exit** to terminate the entire workflow.
- **empty** to take no action.
- **save** to save content, e.g. value of variables, result of queries, etc., produced by the workflow. It is mainly used for eventual results retrieval by a user.

To enable the description of more complex structures, the following *structured activities* are provided:

- **sequence** to define an execution order.
- **flow** to define parallel execution.
- **for_each** to iterate over the results of a **query** activity or over an integer-based counter.
- **if** for conditional execution.

These structured activities can be composed and nested with each other. Furthermore, given a set of activities contained within the same **flow** or **sequence**, the execution order can further be controlled through control **links**, which allow the definition of dependencies

between two activities. A target activity may only start when the source activity associated to it in the **link** has ended. Activities can be connected through links to form directed acyclic graphs.

SCoLa allows one to interact with the services only in two predefined ways: (1) starting an analysis with the **invoke** activity and (2) querying the results of an analysis with the **query** activity. This major simplification is the main difference with WS-BPEL; it is possible because of *SOFAS* services' uniform interfaces. Moreover, from *SCoLa*'s perspective all service calls are considered strictly synchronous and every service replies as soon as it is invoked. Thus, no wait or callback mechanism needs to be defined by the user. We opted to support asynchronicity using polling rather than callback as, in our opinion, it conforms more to REST, while callbacks belong more to an RPC approach. Using polling human users and all sort of applications can use the services in a simple and straightforward way, without having to implement any callback functionality. Moreover, we can build them regardless of how they will be invoked and by whom.

At last, the language does not have any explicit exception handling. All this allowed us to greatly simplify the language without losing much expressiveness. Exceptions, asynchronicity and other low level concepts such as logging and monitoring are supported, but they are simply hidden from the user. They are always handled in the same, standard way and automatically weaved into workflows when they are translated into executable form by *SOFAS' Services Composer*. While extremely important for the actual success of a workflow execution, we deemed all those concepts irrelevant to the user in the case of software analysis composition. Therefore we saw no real benefit in allowing the fine tuning of them by a user and preferred simplicity and conciseness.

However, this does not mean that exceptions and errors are completely hidden from the users, only their handling. If workflows fail, the system will take care of tracking which service(s) failed, why it happened and communicate that to the user, through its automatic exception handling. For example, the most common errors are usually caused by wrong or incomplete input to one or more services. In such cases the *SOFAS' Services Composer* will automatically retrieve the erroneous input from the failed service(s) state and report that to the user.

As in WS-BPEL, *SCoLa*'s workflows can either be executable or abstract. Executable workflows can be submitted 'as-is' to the *SOFAS' Services Composer* for execution as they contain all the necessary information. Abstract workflows, on the other hand, are only partially specified and are not intended to be executed. They hide some of the required concrete operational details, i.e. the value of workflow variables, of some of the input to be

fed to the services, or even the services themselves. Calls to specific analysis services can be substituted with calls to abstract services. They are called abstract as they only exist in their WADL description and describe the features that are common to all the services belonging to a specific category of our analysis taxonomy. They are blueprints that all services belonging to the associated categories need to follow. In our case, the ontologies that their results and input (in case they consume data coming from other analyses) need to conform to. For example, our *GIT version history service*, can also be substituted and described by the generic, abstract *version history service* which defines the pattern and structure that any service extracting the history of a version control repository need to follow. That is, no mandatory, standard input, but output following the version control history ontology defined in our *SEON*. In terms of a *SCoLa* workflow, a call to any of these abstract 'parent' services will be the same as a call to any of its concrete children. It will simply be slightly simplified, as any service specific input is omitted and will only be added once the abstract workflow is instantiated with concrete services. However, this is enough to define a valid workflow, as all the necessary data flow and connections between the services involved is specified by the ontologies describing what they produce and consume. Even when using concrete services, attributes and workflow variables can be completely omitted or their actual value can be left undefined by using what is called an *opaque value*. They will be given the value *##opaque* which prior to execution would need to be substituted with some valid, real values given by the user instantiating the workflow.

Abstract workflows are useful to define a wide array of templates or *blueprints* at different level of abstraction. By using concrete services with *opaque values* for some of their input attributes, it is possible to define workflows that can be easily reused to analyze different projects using the same analyses. The use of abstract services, on the other hand, allows to write more generic workflows that can then be instantiated with different concrete analyses depending on the need of the moment or to have different results. For example, a workflow calculating the code clones of every release of a software project could be defined using the abstract *code clones* and *version history* services. At instantiation time the user would then need to: (1) pick the service working on the needed version control system,(2) pick the code clones detector using the desired strategy and (3) pass them the necessary settings, i.e. the URL to the repository to analyze and clone detection specific options (e.g. the tokens size). The identification and development of such abstract workflows is not addressed in this work. However, in our opinion it is a very relevant topic that deserves being investigated in detail in the future.

6.3.2 Workflow validation

All service input is either in the form of strings or files. The latter being files to be analyzed (e.g. source code) by the specific service; the former being both options of the analysis (*e.g.*, URL of the version control repository to extract the history from) and the URL to data produced by other services (a.k.a. their output) to be fed to the service. All the analysis options are provided manually by the user during the composition. The syntactical correctness is checked and enforced using constraints defined in the web service description using XML schema restrictions. Restrictions are used to declare acceptable values of XML elements and attributes. For example, limit the valid content of an element to some predefined series of numbers or letters. As of now, no check is done when the input data is in the form of files. It is up to the user to provide valid, accepted files.

What is vital for the validity and correctness of *SCoLa* workflows is that the data flow between services is also semantically correct. That is, given any source service and the target services depending on its output, the source produces exactly the data the targets need and represents it in the right format. That means, for example, given one of our *version history services* and the *version history metrics service*, ensuring that the results of the former that feed into the latter is a version control history and is described using the proper ontology; in our case one of the *SEON* ontologies previously introduced in Section 6.2.2.

The validation and verification of service workflows has been addressed from different perspectives in several papers: from formal, model-based, verification of workflows [FUMK03, BBG+07] to data validation [HBA08, XQW+10]. However, all these approaches focus on classic "big web services". So far, these issues, and in particular semantical correctness and validation, have not been addressed for RESTful services yet. To mitigate this problem, we devised an ad-hoc, light-weight, validation technique based on the WADL service description. Taking inspiration from SAWSDL (Semantic Annotations for WSDL) [FL07] and SA-REST [LGS07], we expanded the WADL description so that any input and output of a service method may be annotated with a URI to an ontology, or an ontology class, that logically represents it, as shown in Figure 6.2.

A connection between two services in a *SCoLa* workflow will be deemed semantically valid only if, based on their WADL descriptions, the output provided by the source service and the input of the target service to which it is linked (using a SCoLa control **link**) represent the same particular ontology or the same ontology concept. Obviously, only executable workflows can be fully validated. Abstract workflows can be semantically

```
<?xml version="1.0" encoding="UTF-8">
<application xmlns="http://evolizer.org/wadl"
             xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <resources base="http://habanero.ifi.uzh.ch/releaseFamix/">
    <resource path="/analyses">
      <method id="createHistoryWithForm" name="POST">
                The method to start the analysis
        <request>
          <representation mediaType="application/x-www-form-urlencoded">
            <param name="name" style="query" type="xs:string">
                              The name of the new analysis
            </param>
1           <param name="url"
                    style="query"
                    type="xs:string"
                    ontologyClass="http://se-on.org/ontologies/domain-specific/2012/02/history.owl#Release"/>
                    The url of the release to extract the FAMIX model of
            </param>
          </representation>
        </request>
2       <response>
          <representation mediaType="text/html"
                          ontology="http://se-on.org/ontologies/domain-specific/2012/02/code.owl"/>
        </response>
      </method>
      . . . .
      <resource path="{name}">
        . . . .
      </resource>
    </resource>
  </resources>
</application>
```

Figure 6.2: Snippet of the release meta-model service WADL description. The service is declared as requiring a release of a history ontology instance (point 1) and returning as output in the form of a static source code structure meta-model (point 2).

validated, as they have to declare all the links between the different services. A full semantic and syntactic validation is not possible as, being abstract, some variables and attributes are omitted or given *opaque values*.

This customization of WADL is also useful in guiding the user in the creation of workflows. In fact, given a specific service, based on its description and on the results it is declared to require and produce, it is then trivial to automatically fetch from the *Services Catalog* all the ones that produce and consume that particular data and propose them to the user. We will see this in more details in the next Section.

An important note to be made is that our goal was not to provide a full fledged, comprehensive, validation approach as the ones already proposed for classic web services. We developed a light-weight, yet rich enough for our needs, technique tailored to the very specific nature and structure of *SOFAS* services. It was not the main focus of our work, but rather a means to achieve our goal of a flexible framework to compose analyses.

We opted for WADL instead of the more widely used WSDL to describe our services for several reasons. First of all, it is the *de facto* standard to describe RESTful web services. Even though it is not widely used yet, it is, in our opinion, the best way to describe RESTful

web services. In fact, it was exactly devised to describe web applications that use the HTTP protocol to communicate in a simple yet effective manner. Even a complicated business application is described as basic operations (PUT, GET, POST, DELETE, etc.) on the resources that comprise the application's state (in our case an analysis). On the other hand, WSDL was designed to describe all sort of service interfaces using just about any protocol imaginable. This makes it a much more complex and expressive language. WSDL 2.0 even defines special HTTP bindings to describe HTTP applications in a very rich way. However, such richness comes at the cost of increased complexity. In our opinion, this added complexity would have been unnecessary for our purpose and would have outweighed the gained expressiveness. In fact, we are able to successfully describe our services with WADL without compromising accuracy and detail. In conclusion, WADL better fits our needs, as throughout our approach, we favor, when possible, simplicity and ease of use over expressiveness and feature richness.

An important note to be made is that our goal was not to provide a full fledged, comprehensive, validation approach as the ones already proposed for classic web services. We developed a light-weight–yet rich enough for our needs–technique tailored to the very specific nature and structure of *SOFAS* services. It was not the main focus of our work, but rather a means to achieve our goal of a flexible framework to compose analyses.

6.3.3 Workflow creation and execution

SCoLa workflows are executed by the *SOFAS Services Composer*, which translates them into a concrete and executable form. They can be composed and submitted for execution in two ways: using its REST API or the *SA-B* UI. In the first case, the bare XML-based description has to be manually compiled and submitted for execution by the user to the *Services Composer* through its REST API. This option is useful for tools to automatically compose and submit their own workflows, but not for a human user. In fact it does not exploit all the benefits of *SOFAS*' guided analysis composition and the system in general. The burden of finding the right analyses, composing them in the right way, knowing the composition language, etc. is all on the user. The *SA-B* UI offers an intuitive graphical "boxes and arrows" way to compose workflows, as shown in Figure 6.3. This second solution exploits *SOFAS* at its fullest, showcasing the benefits of such an integrated approach. Through its Ajax-based interface, the user can find and pick the analyses needed from the catalog browser, connect them together and provide all the necessary input all at once in the workflow editor.

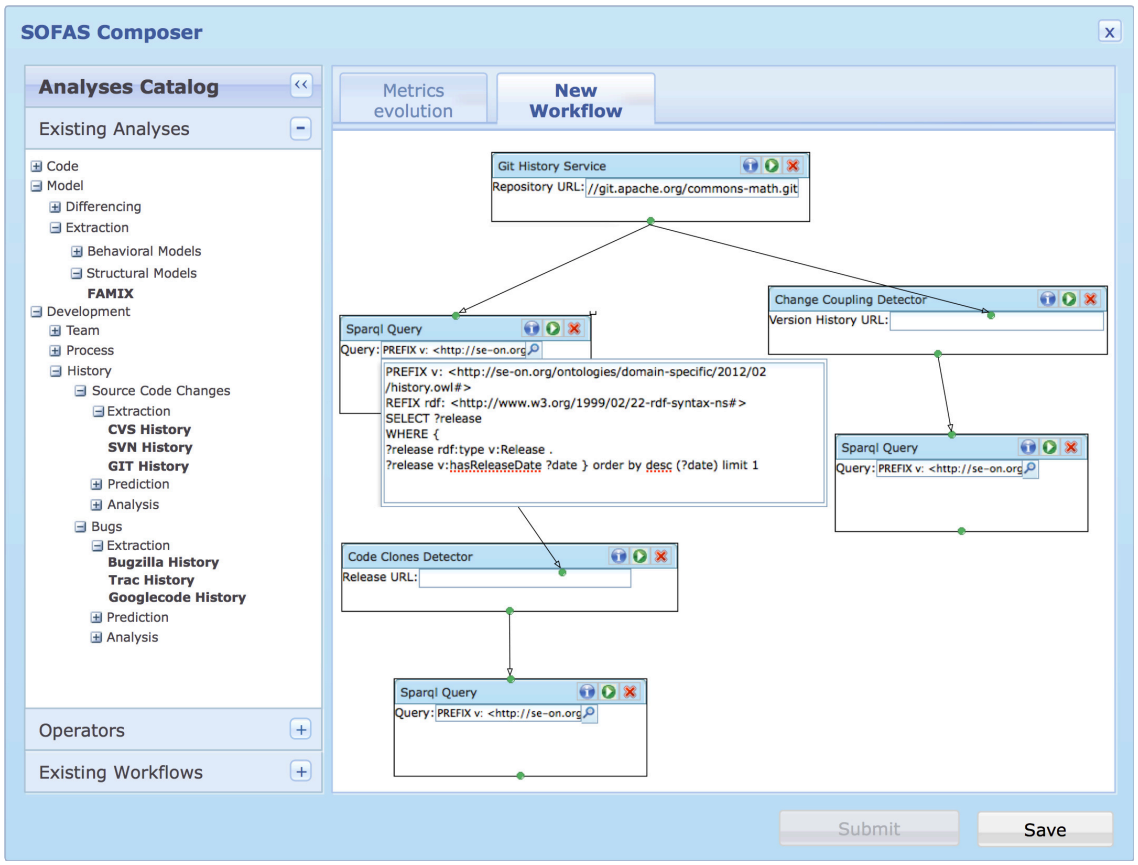

Figure 6.3: Screenshot of workflow composer of the *SA-B* web UI while being used to define the Hotspot workflow described in Section 6.5.1

Moreover, the user is guided by the system into this composition. Once a service is picked, all the ones that consume its output or that supply data needed by it are suggested. This is possible because of the custom annotations added to the services WADL descriptions we previously explained. The moment a service is selected, the composer automatically browses the catalog to fetch all the possible compatible/related services to suggest. Furthermore, thanks to this, workflows are validated as they are composed, in real time and wrong combinations are exposed as soon as they are created. Using the workflow in Figure 6.3 as an example, if the *change coupling detector service* is selected first, all the *version control history services* will be suggested as they produce data needed by it. By using this UI, not only the actual analyses but also their REST API are hidden from the user. She would just need some very basic information about the system to study, e.g. version control repository URL, source code, etc. and all the technicalities will be hidden

behind the intuitive and simple boxes and arrows interface.

Workflows can be saved for future reuse or modification. On top of that, the *SA-B* builds and manages them as RESTful services, providing the exact same interface as all the other *SOFAS* services: their required input is the combination of input required by every single service with the exception of the one provided by other services in the workflow. This means that they can then be used as any another analysis service and combined, as atomic entities, with other services and workflows. *SOFAS* comes with some predefined workflows, called *analysis blueprints*, representing some of what we think are the most common and useful analyses to shed light on a software project and its evolution. These analyses are based on some of the most common software evolution analysis studies published in software engineering conferences (*e.g.*,, MSR, ICSE, FSE, etc.) and on software evolution analysis needs originating from concrete industrial case studies.

6.4 Software Analysis Broker

The *SA-B* acts as a "layer" between the services and the users, so that they do not have to interact directly with the raw services. It plays a vital role in facilitating the use of the services in an effective and meaningful way. In particular the composition and execution of the *SCoLa* workflows we just introduced in the previous section. Four main components constitute the *SA-B*: the *Services Catalog*, a series of management tools, the *Services Composer*, and a user interface.

6.4.1 Services Catalog

The *Services Catalog* stores and classifies all the registered analysis services so that a user can discover services, invoke them, and fetch the results. We developed a software analysis taxonomy to systematically classify existing and future services. This taxonomy divides the possible analyses into three main categories: development process, underlying models, and source code. For more details we refer to the *SOFAS* website[10].

As the data produced by the analyses, also this taxonomy is defined as an OWL ontology. This allows us to have a very complex and rich service classification. Furthermore, SPARQL can be used to query the catalog and fetch specific services. With it, services

[10]http://www.ifi.uzh.ch/seal/research/tools/sofas.html

can be queried based on what categories they belong to, on any of their attributes, on the attributes of any of the categories they belong to, etc.

6.4.2 User Interface

The UI is the actual access point to the *SA-B*. It consists of a web GUI, meant for human users and a series of RESTful service endpoints to be (semi)-automatically used by applications. Through it, the user can browse the *Services Catalog* to find the needed analyses, compose them and eventually run them. The user can also pick from some already predefined combinations of analysis services provided as high level analyses workflows (called *analysis blueprints*). Services can be combined into *SCoLa* workflows in a intuitive, high level and graphical "pipe and filter" fashion, as we already mentioned in Section 6.3.3.

6.4.3 Services Composer

This component takes care of translating the workflows defined through the UI into actual, executable ones and execute them. Having the composition definition and the actual composition language decoupled, allows the user to compose services in an intuitive way, hiding the complexity and technicalities of the actual composition and orchestration. Moreover, calls to additional services, such as the ones described in Section 6.4.4 can be automatically weaved into a user-defined workflow.

6.4.4 Services Management Tools

A workflow is not just a mere collection of services, called one after another. In particular, this holds when long running, asynchronous web services are involved. In order to effectively execute it, every single service needs to be logged and monitored to check if it is up and running, if it is in an erroneous state and why, if it completed a required operation, etc. We implemented a series of services that take care of implementing that as services. Calls to them can be easily and automatically weaved into a user-defined workflow by the *Services Composer*.

6.5 Applications of Software Analysis Composition

In the following, we present two applications of *SOFAS* analysis composition. The first one is the use of the *SA-B* web UI by human users to define analysis workflows to answer specific software evolution questions. The second one is the use of the *SA-B* RESTful endpoints by a tool called *Software Evolution Perspectives*, to execute a workflow for extracting and visualizing evolutionary data in various perspectives.

6.5.1 Investigating Evolution Anomalies with Analysis Workflows

As a use case for this first type of application of *SOFAS*, we show how we can answer the question *"Which are the hotspots and evolution anomalies for a project?"* by composing a specific workflow using the *SA-B* web UI. Figure 6.3 shows the UI being used to compose this very workflow. This question and the associated workflow originate from a concrete need we encountered while performing a software quality audit of a commercial software. This software, which we will call Andromeda (due to confidentiality obligations we cannot disclose its real name), is a mission critical system in the domain of facility management for monitoring and maintaining buildings.

We consider a hotspot any source code entity (file, class, method, etc.) that is out of the norm according to different heuristics. Such anomalies have been proven to have a negative effect on software quality, often leading to faults and defects, code brittleness and maintainability issues [DLR09, BBM96]. Different strategies have been devised to spot them to then support and drive the reengineering process. Some used empirically validated object-oriented source code metrics [GFS05, BBM96], others a combination of them to find more complex and higher-level problems known as "code disharmonies" [LM05]; others used change couplings extracted from the project revision history [DLR09], etc. All these approaches are valid, however, they do not necessarily find the same hotspots. For example, metrics-based solutions would mainly find code quality related anomalies, while an analysis working on the change couplings between files would find more high level issues such as cross cutting concerns.

With *SOFAS* we can compose workflows that combine some of these different strategies to find a broader spectrum of hotspots and to also find "super hotspots": entities that

present anomalies found with several of the strategies used. The workflow we present here aims to answer the main, generic question by answering and combining four, more specific sub-questions:

1. *Which entities have high or abnormal values of known code quality metrics?*
2. *Which entities present specific code smells?*
3. *Which entities have a high change coupling?*
4. *Which entities have a lot of copied code (code clones)?*

The results are then aggregated to find classes that exhibit all "symptoms," which is the final question: *which entities are super hotspots?* Figure 6.4 provides an overview of it. As first step, the version history of the project to study is extracted from the associated repository, along with necessary information about the repository. The data produced by this version history service is then fed into a service calculating the project's change couplings (how frequently a class has been changed together with other classes over the evolution of a project).

The *SCoLa* code snippet in Figure 6.5 shows how this connection is defined in the language. The two services in the snippet, as the workflow itself, are abstract. This is because the version history service has to be picked at runtime, according to the specific repository used by the project. The change coupling is also left to be instantiated at runtime to further fine tune the analysis with a selection of services implementing different detection strategies. The results of the change couplings service are further refined with a SPARQL query to extract the 10 most significant classes found and answer the analysis question 3. These are the classes with the highest number of coupled classes (NOCC) and sum of coupling (SOC) metrics. The data produced by the version history service is also fed to a SPARQL query to find the most recent release. This is fed to a service extracting all its code clones and to a service extracting its static source code structure model.

The results of the code clones analysis are fed to a SPARQL query to extract the 10 classes with the highest number of clones and thus answer the analysis question 4. The source code model extracted is then fed to two metrics services. One extracting the most common object oriented metrics and the other one extracting LOC and control flow-related metrics: McCabe's cyclomatic complexity [McC76a], weighted methods per class (WMC) [CK94], etc. The results are input to a service that aggregates them and finds code disharmonies, as proposed by Lanza *et al.* [LM05]. This answers analysis question 2.

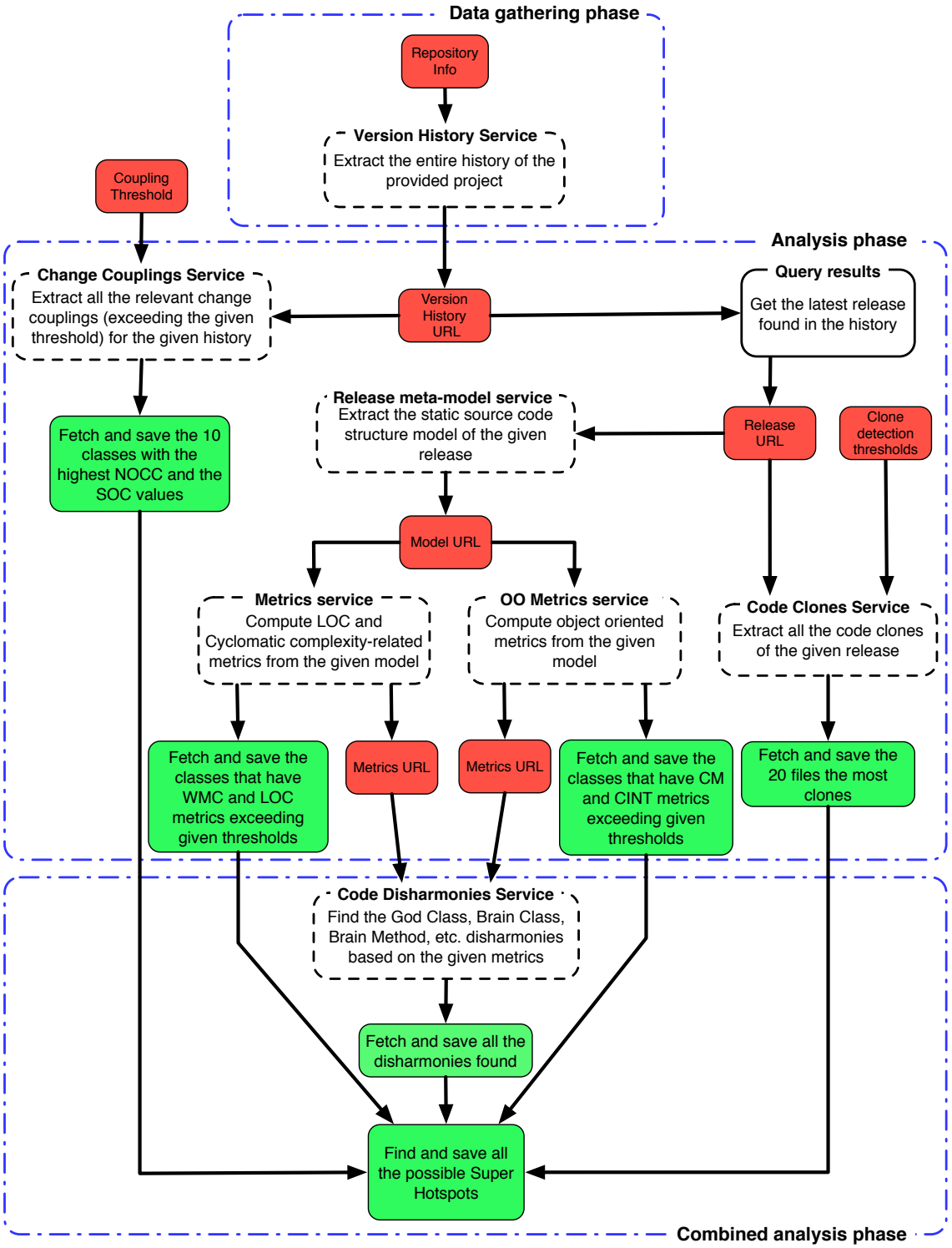

Figure 6.4: Overall view of the Hotspots workflow.

```
<links>
      <link name="versionHistory_to_changeCoupling"/>
      <link name="versionHistory_to_fetchReleaseQuery"/>
</links>
<sequence>
      <invoke analysis="http://habanero.ifi.uzh.ch/sofas/abstractVersionHistory"
           inputVariable="repositoryData"
          outputVariable="historyUrl">
        <sources>
            <source linkName="versionHistory_to_changeCoupling"/>
            <source linkName="versionHistory_to_fetchReleaseQuery"/>
        </sources>
     </invoke>
     <assign>
        <copy>
            <from>$historyUrl</from>
            <to>$changeCouplingInput/customerInfo</to>
        </copy>
        <copy>
            <from>$historyUrl</from>
            <to>$inputUrl</to>
        </copy>
    </assign>
    <flow>
        <invoke analysis="http://habanero.ifi.uzh.ch/sofas/abstractChangeCoupling"
            inputVariable="changeCouplingInput">
            <targets>
                <target linkName="versionHistory_to_changeCoupling"/>
            </targets>
        </invoke>
        <query sparqlQuery="SELECT ?x WHERE ?x rdf:type v:Release . . . . . ."
             inputVariable="inputUrl"
             outputVariable="lastReleaseUrl">
             <targets>
                 <target linkName="versionHistory_to_fetchReleaseQuery"/>
             </targets>
        </query>
    </flow>
</sequence>
```

Figure 6.5: A snippet of the *SCoLa* definition of the Hotspots workflow.

The results are also fed to SPARQL queries to extract the metrics that in studies of Basili *et al.* [BBM96] or Gyimothy *et al.* [GFS05] have been found to be relevant for defect prediction. So we can answer analysis question 1. Finally, all the results of these four sub-questions are aggregated to find any possible "super hotspot." All the results are also saved, so that the user can eventually fetch and analyze them upon the workflow completion. The *SCoLa* snippet in Figure 6.6 shows how this aggregation is defined. Notice, however, that

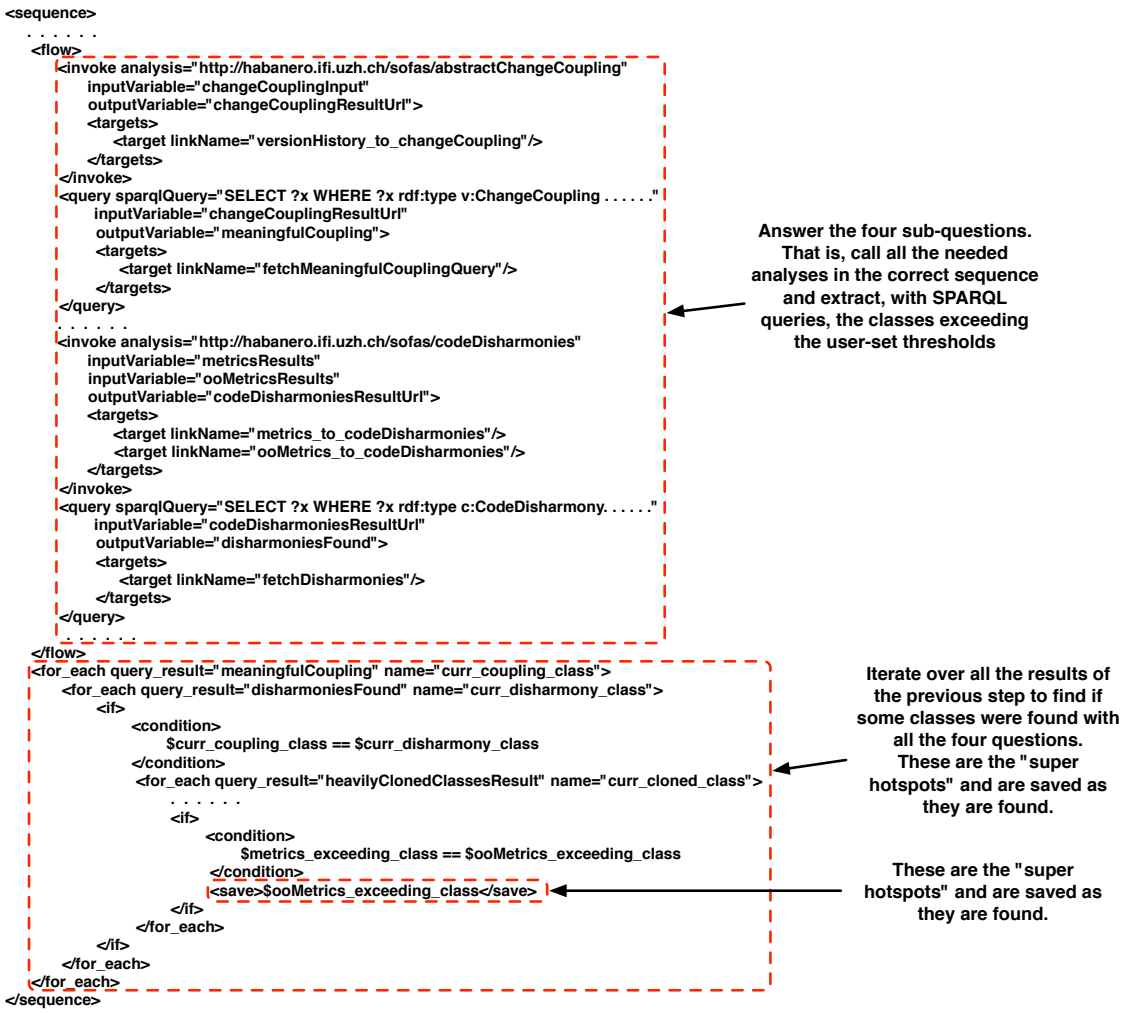

Figure 6.6: A snippet of the *SCoLa* definition of the Hotspots workflow showing how results are aggregated and saved.

the services do not share the actual data, which in the case of versioning data could even be in the range of gigabytes. Only the URL is, and through that, data can be selectively

fetched is using *SOFAS'* uniform RESTful interface (as described in Section 6.2). Thus the traffic between the services is kept to a minimum. It will be up the individual services to get all the necessary data needed.

This workflow has been run for the Andromeda system during a software quality assessment process. In addition to that, it has been ran as a use case validation for some of the most popular Apache Commons[11] projects: commonsValidator, commonsTransaction, commonsMath, commonsLang, commonsIo, commonsCollections, commonsCodec and commonsCli.

Tables **??** summarize the results for all these projects, grouped by the analysis questions they answer. Please note, that due to space limitations, we are showing only the 5 top most relevant files for each analysis question for the analyzed Apache projects analyzed. In some cases, the relevant entities exceeding the required threshold where even less than that (e.g. for Commons Collections and Cli). On the other hand, we show the entire result set for Andromeda. The classes underlined are the "super hotspots" (if any were found). Andromeda's classes have been renamed due to confidentiality needs.

6.5.2 Software Evolution Perspectives with Analysis Workflows

Analysis workflows such as the one presented in the previous section are extremely valuable in answering specific evolution analysis questions and in singling out, unequivocally, noteworthy entities. However, when used by themselves, they lack the capability to fulfill broader and more open-ended information needs. For example, giving an overall view of the evolution or the current state of a software project or to show trends of specific, critical metrics. They can still provide all the information needed to fulfill those needs but, in this case, human interpretation is heavily needed to put everything into context and draw meaningful conclusions. Our web application *Software Evolution Perspectives* aims exactly at filling this gap.

The main purpose of this tool is to give software evolution and quality analysts a detailed and intuitive overview on the quality of a software project and its history. This is achieved through the use and combination of different "perspectives", focusing on different aspects of the software analyzed. Every perspective offers different interactive visualizations of the aspect addressed, along with automatically generated considerations

[11] http://commons.apache.org

Project	Which entities have a high change coupling?	Which entities have a lot of duplicate code (code clones)?
Andromeda	Eve.java Fdd.java Con.java Por.java Bas.java	Eve.java Con.java Por.java Use.java
Commons Validator	ValidatorResources.java Validator.java Field.java Form.java ValidatorAction.java	Field.java DateValidator.java EmailValidator.java UrlValidator.java ValidatorAction.java
Commons Math	EmpiricalDistributionImpl.java GammaDistributionImpl.java RealMatrixImpl.java AbstractContinuousDistribution.java ExponentialDistributionImpl.java	ComposableFunction.java MathUtils.java OpenIntToDoubleHashMap.java AbstractRealMatrix.java RealMatrixImpl.java
Commons IO	AndFileFilter.java OrFileFilter.java PrefixFileFilter.java NameFileFilter.java SuffixFileFilter.java	FileUtils.java NameFileFilter.java IOUtils.java FileWriterWithEncoding.java Tailer.java
Commons Codec	Base64Test.java Base64.java RefinedSoundex.java Metaphone.java URLCodec.java	DoubleMetaphone.java RefinedSoundex.java QuotedPrintableCodec.java URLCodec.java Hex.java
Commons Cli	Option.java CommandLine.java Options.java PosixParser.java GnuParser.java	OptionBuilder.java
Commons Collections	CollectionUtils.java MapUtils.java ListUtils.java BufferUtils.java SetUtils.java	
Commons Transaction	GenericLock.java GenericLockManager.java FileResourceManager.java LockManager.java ResourceManager.java	FileResourceManager.java TransactionalMapWrapper.java GenericLockManager.java AbstractXAResource.java FileHelper.java

Table 6.1: Project Hotspots workflow answers to the first and second question

Project	Which entities present specific code smells?	Which entities have high or abnormal values of known code quality metrics?
Andromeda	Eve.java Fdd.java Com.java Obj.java Poi.java Con.java Obi.java	Cons.java Eve.java Con.java Com.java Evn.java Jta.java Obi.java Rea.java Wat.java Dow.java
Commons Validator	Field.java	Field.java
Commons Math	TransformerMap.java	StatUtils.java DefaultTransformer.java
Commons IO		NameFileFilter.java IOUtils.java XmlStreamReader.java
Commons Codec		
Commons Cli		
Commons Collections		
Commons Transaction	GenericLock.java	FileResourceManager.java

Table 6.2: Project Hotspots workflow answers to the third and fourth question

about it. These considerations are used to better explain the different visualizations and to put them into a software quality context. Moreover, they also explain, for example, why specific results and values found are good/bad for the quality of the software analyzed. The perspectives offered so far are:

Metrics perspective. It addresses the visualization and interpretation of the metrics calculated for every release of the analyzed software project–and the code smells detected based on those metrics. These visualizations include:

- The metrics pyramid [LM05] of every release.
- Interactive, navigable kiviat diagrams of all the packages and classes of the system, as proposed by Pinzger et al. [PGFL05].
- The evolution, over time, of the most important project-wide metrics (e.g. LOC, average cyclomatic complexity, etc.).
- A browsable list of all the disharmonies and the code entities (classes and methods) that exhibit them.
- A navigable, interactive treemap of every release of the entire system, with the exceptional entities highlighted so that they can be quickly pinpointed and studied. Exceptional entities are packages or classes that either exhibit values of critical metrics above known thresholds or specific code disharmonies.

Figures 6.7 and 6.8 show a small collection of these visualizations.

Project history perspective. It addresses the visualization and interpretation of the history of the project. These visualizations include:

- Interactive, navigable fractals representing the code ownership of every class and package of the project, as proposed by D'Ambros et al. [DLG05].
- Graphs of some of the most commonly used version control metrics/statistics(i.e. distribution of commits between developers, code churn, etc.).
- Graphs of some of the most commonly used issue tracking metrics/statistics(i.e. bugs open/closed per month, distribution of bugs by different attributes, etc.).

Two of these visualizations are shown in Figure 6.9.

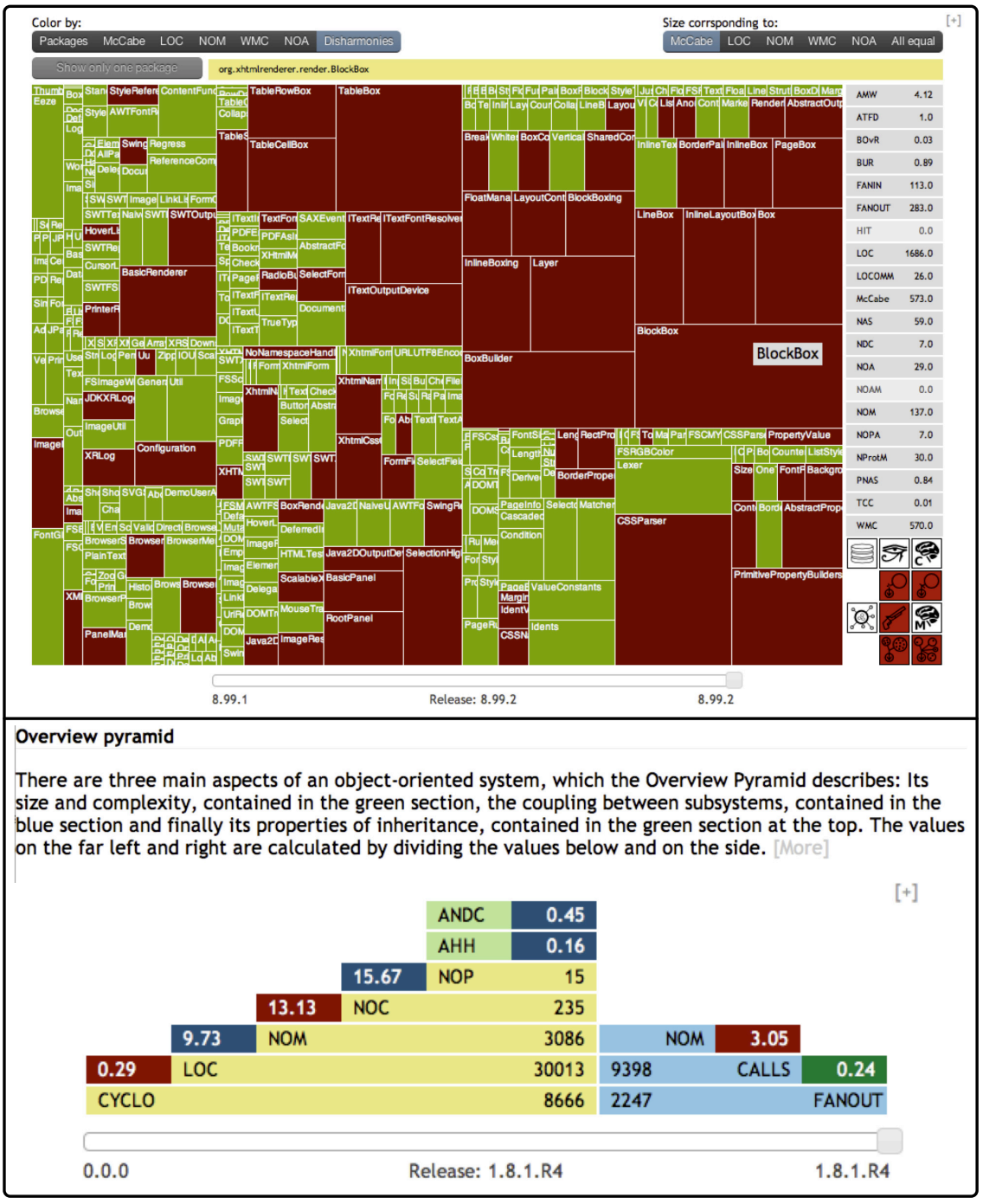

Figure 6.7: A screenshot of a metrics pyramid and an interactive kiviat diagram of the Metrics perspective.

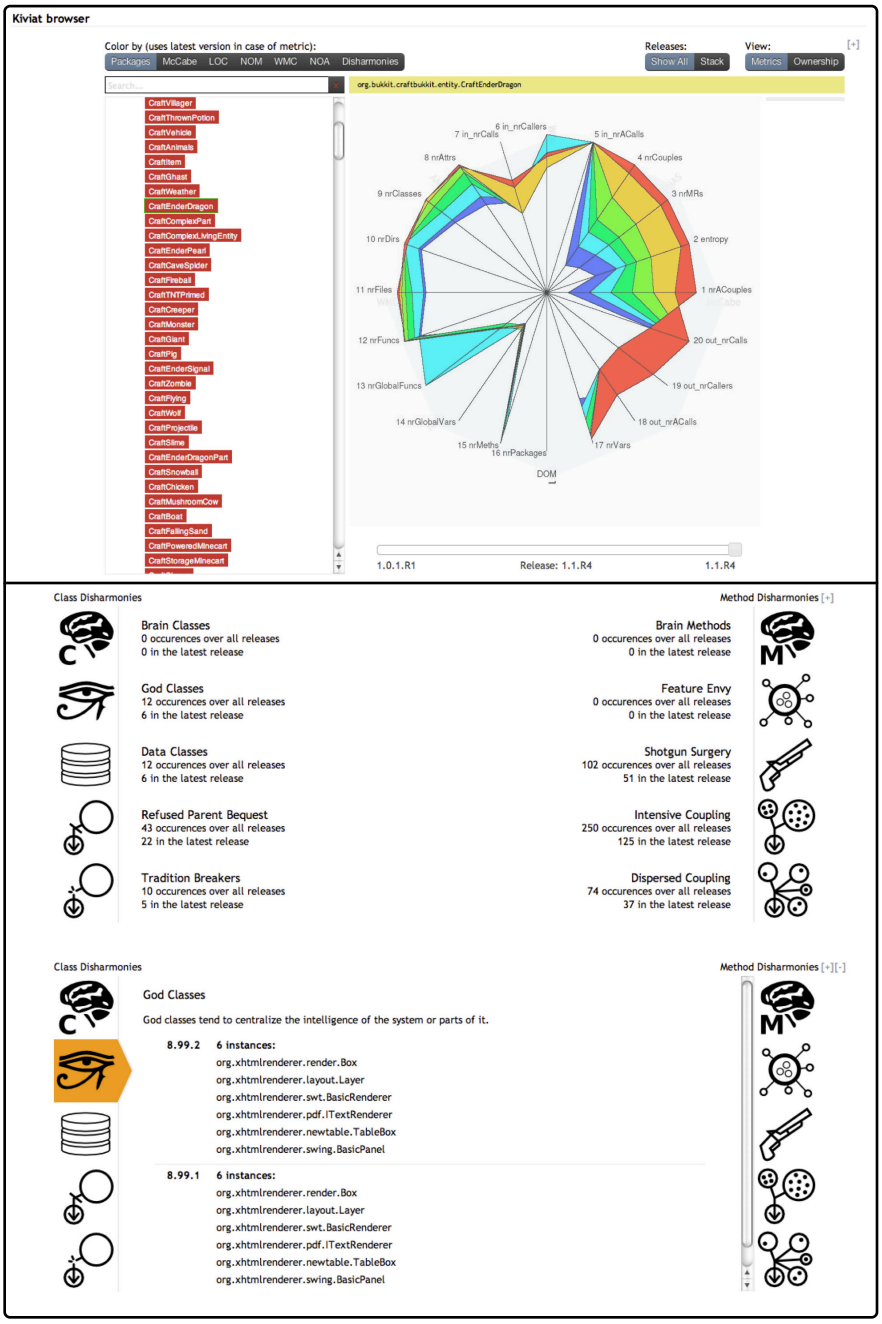

Figure 6.8: A screenshot of the navigable, interactive treemap and the disharmonies list of the Metrics perspective.

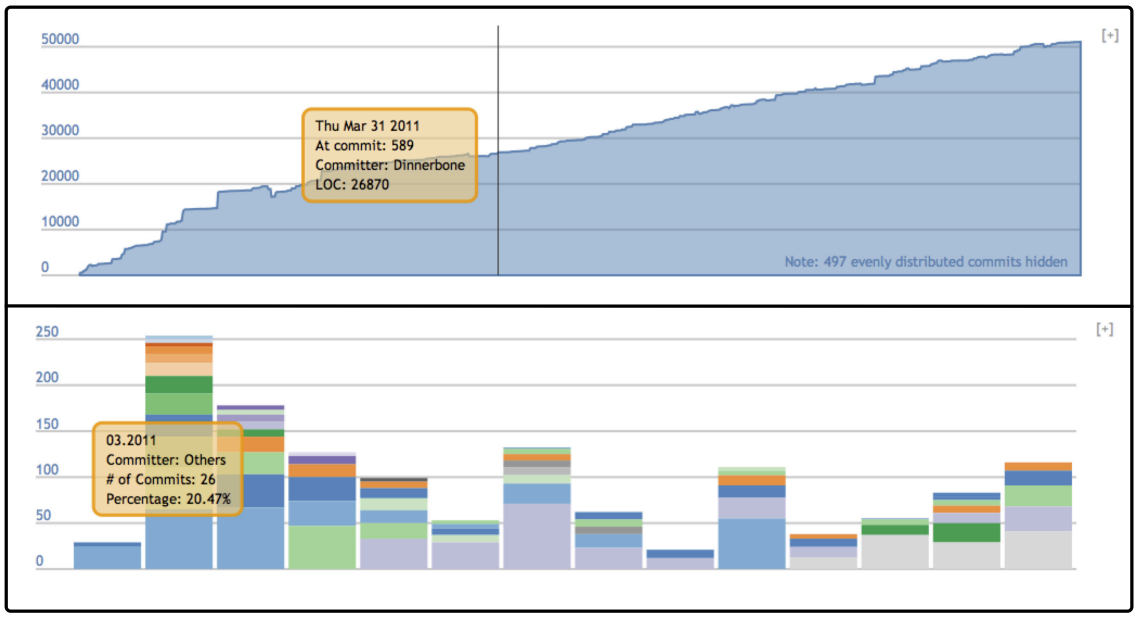

Figure 6.9: Two of the visualizations making up the project history perspective.

Change coupling perspective. It addresses the in-depth visualization and interpretation of the logical coupling between entities (files, source code files, modules, directories, etc.) at different granularity levels, leading to a precise characterization of the system modules in terms of their logical coupling dependencies. It is based on D'Ambros et al.'s Evolution Radar [DLL09].

Fine grained source code changes perspective. It gives a detailed view of all the fine grained source code changes that happened throughout the project history. This perspective provides detailed information on the statement and declaration level changes that is missing in the normal change history available in version control system. The visualizations offered include:

- A navigable, interactive change history of every single file, module or the entire system. Showing, for every commit, its overall significance and detailed information on all the associated fine grained changes.
- Piecharts showing the distribution of these changes over the entire history of a single file, module or the entire system.
- A list of the 10 most significant commits in the project history, with details on which files where changed in it and how (with which type of changes).

In addition to these interactive visualizations, *Software Evolution Perspectives* also offer the automatic creation of a software quality and history report based on some of the most relevant visualizations and analyses used in the different perspectives. This report gives a quick overall assessment of the quality of the analyzed project, its hotspots and how these evolved in time. As for the Hotspots workflow we introduced in Section 6.5.1, this was also used in a real software quality audit project with an industrial partner to analyze the Andromeda system.

Software Evolution Perspectives uses a custom, predefined *SCoLa* workflow to combine and execute all the analyses required to get all the data needed by the different perspectives. Figure 6.10 shows the high-level representation of such workflow. This workflow just combines all the different analyses needed, returning only the URLs to their results. It does not aggregate or work on the results like the one we presented in Section 6.5.1. This is because, as said, the goal is not to answer a specific question, but to get as much information as possible about the quality and history of the analyzed project. This means that *Software Evolution Perspectives* will then, given those result URLs fetch and aggregate all the data needed directly from the analyses providing them.

Another major difference with the use case presented in the previous section is that, in this case, the actual workflow and *SOFAS* itself are hidden from the user. While to answer specific questions, such as the one in Section 6.5.1, the user uses *SA-B* web UI to compose a suitable workflow, in this case, the tool itself will take care of that. The user will only have to supply the URL of the version control and issue tracking repositories of the project to analyze. The tool then, will automatically detect the systems those URLs refer to, e.g. git, svn, bugzilla, trac, etc., compose a suitable workflow and send it to *SOFAS*' *SA-B* for execution, through its RESTful endpoints. Upon workflow completion, the tool will then fetch all the data needed from the analysis themselves, save it and organize it into the different perspectives.

6.6 Related Work

In this section, we briefly outline some of the major existing works related to our approach. In particular, we discuss the use of ontologies in mining software repositories and software evolution, RESTful web services composition, and tools exploiting and combining historical project data for software evolution analysis.

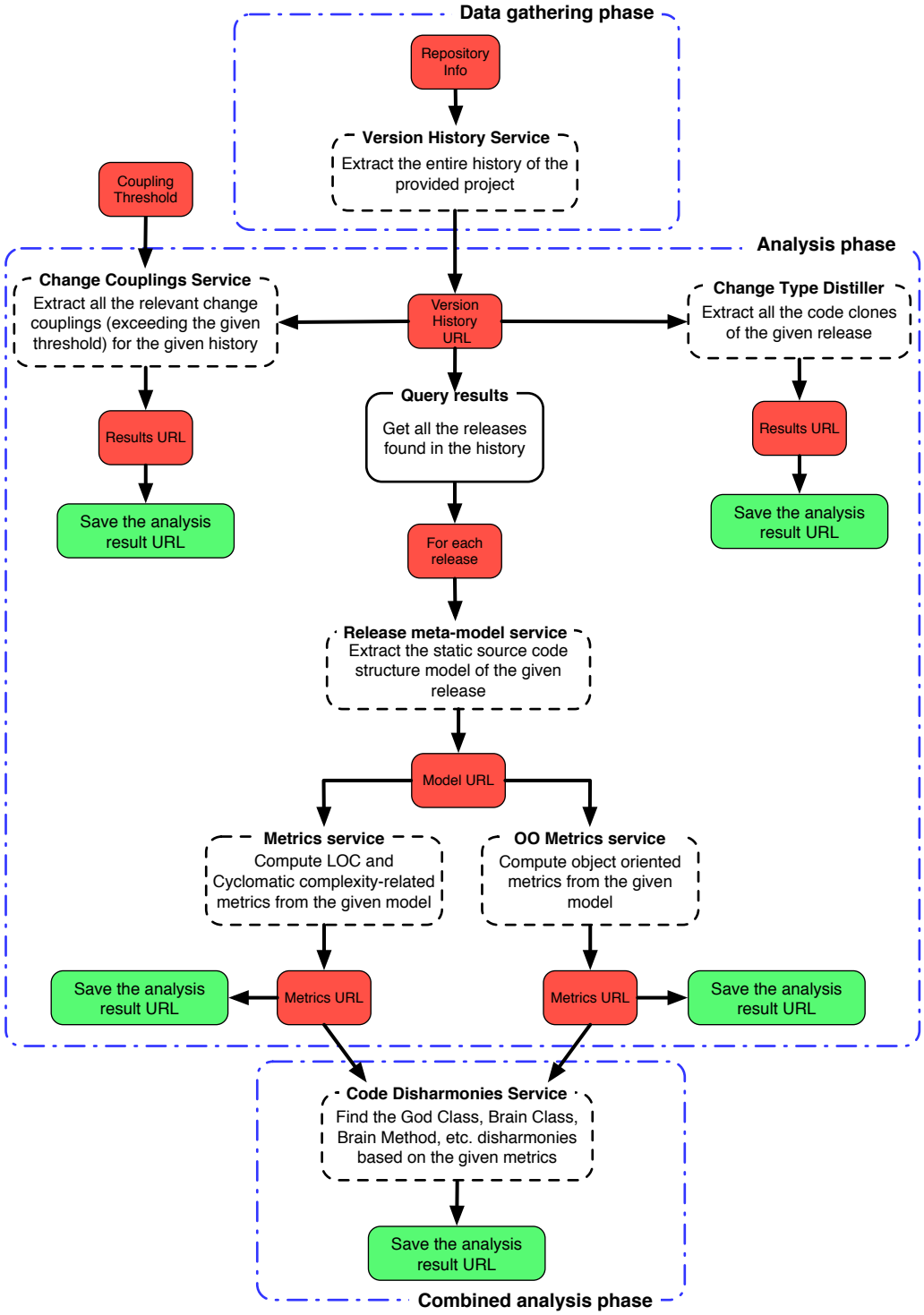

Figure 6.10: Overall view of the workflow used by *Software Evolution Perspectives*.

6.6.1 Ontologies in Software Evolution Analysis

Several researchers have described software evolution artifacts with OWL ontologies. Their approaches integrated different artifact sources to facilitate analysis activities.

Kiefer et al. proposed EvoOnt, a software repository data exchange format based on OWL [KBT07]. EvoOnt is heavily inspired by Evolizer's [GFP09] data models and is made up of three sub-ontologies: a software ontology model, a bug ontology model and a version ontology model. The authors used a modified version of SPARQL to detect bad code smells, calculate metrics, and to extract data for visualizing changes in code over time. In an extension to this work, Tappolet et al. [TKB10] replicated several software evolution and analysis experiments from previous Mining Software Repositories Workshops. As a result, they showed they could replicate 75% of those analyses with at most two SPARQL queries.

Iqbal et al. [IUHT09] proposed a *Linked Data Driven Software Development* (LD2SD) methodology, which involves transformation of software repository data into RDF format and then indexing with a semantic indexer. The overall goal was to provide a uniform and central RDF-based access to JIRA bug trackers, Subversion, developer blogs, project mailing lists, etc. Integration between the repositories was achieved with *Semantic Pipes*, an RDF-based mashup technology. The results were finally injected into the bug tracker web page, to provide developers with additional, context-related information.

We share with these works the idea of describing varied software repository data with interlinked ontologies. However, none of them organize their ontologies in consecutive layers of abstractions with clear representational purpose, as we did in *SEON*. Moreover, their main goal is to use these ontologies to make the implicit links between different software artifacts explicit and facilitate the use of such information. While that is also one of our goals, in addition to that, we also use ontologies to promote easier information sharing between analyses and to build more complex and composite analyses on top of this core. At last, these approaches only present a proof of concept of the usefulness of ontologies. A generic framework where data about software artifacts can be automatically collected, analyzed and queried for several purposes is still missing.

6.6.2 RESTful Webservice Composition

Web service composition has been throughly addressed–and it still is–in the major software engineering conferences and in the more specific ICSOC and ICWS. A state of the art of

it is not in the scope of this paper and our work in general. On the other hand, the issue of composing RESTful web services is highly related to our work. Traditionally, these services have been used in a much different context than the traditional SOAP RPC-based ones. In particular, they have been used as standalone web application or manually, ad-hoc combined with all sort of other services in web 2.0 mashups. The real need for a standard RESTful web service description language and a structured and methodical combination technique has not really come up yet.

Pautasso [Pau08] and Mandel [Man08] both proposed the use of WSDL to describe RESTful web services to then facilitate their composition with other similar services and with classic "big services". Pautasso also introduced, in the same work, an extension to WS-BPEL to natively support REST, without the need of a WSDL bridge.

Other works, such as [Pau09, ZD09, MSW09], took a more radical approach, proposing the use of new ad-hoc tools [Pau09] or languages [ZD09, MSW09] to describe and compose these services. None of these have really gained much ground or have been used outside theoretical case studies.

Given this situation, it comes as no surprise that more high level concepts as workflow validation or semantic description of web services has not been addressed yet, apart from the recently proposed SA-REST [LGS07]. Oddly enough, so far no solution has exploited the already existing WADL, even though it can almost be considered a de-facto standard. On the other hand, we decided to base our solution on WADL as we deemed it mature and expressive enough for our needs. Moreover, it allowed us to use an already existing language without having to define one of our own, which in the end would have had very similar structure and rationale.

6.6.3 Software Evolution Analysis Composition

There is an abundance of research works and tools exploiting software project data for historical software analysis. The majority of them exploit the source code change history or bug history to study the dynamics underlying software evolution. Only a few address the combination data coming from different analyses and sources.

Systems such as Kenyon [BWKG05], Evolizer [GFP09] and softChange [Ger04] combine source code change history, bug history and additional analyses such as, for example, fine grained source code change extraction, source code meta-model reconstruction, etc. All these approaches rely on their own ad-hoc developed tools and techniques and none target the issue of using and composing different, independent analyses. Furthermore,

they don't allow the user to combine the analyses into custom combinations. All the supported ones are created beforehand and hardcoded into the tools. At last, all of these approaches are tool-based and none addressed the issue of using web services–or similar technologies–to support and facilitate the analysis usage and composition. OASIS [JC05], by Jin and Cordy, has been so far the only attempt to find a solution to most of these issues.

We share with them the overall concept, but at the same time, the two approaches have many differences due to their partially distinct goals. Their objective was to allow an analysis available in one tool to use the fact-base of another one in a very simple way. For this reason, they used a domain ontology just to describe the set of representational concepts that the different tools to be integrated require and support. On the other hand, we exploit ontologies on a much broader scale: to catalog and describe the services, to represent and standardize their input and output accordingly to the type of analysis offered, to semantically link different results, to perform (semi)-automatic reasoning on them and, at last, to support the combination of different analyses. In our opinion, this last point is really the most novel and useful feature of our work. At last, their work only sketched the overall rationale of the approach without going into details on how the proposed architecture was actually implemented and which technologies were used. Based on all these considerations, we can claim that the issue of software analysis composition has not yet been systematically tackled.

6.7 Conclusions

We have investigated the concept of *Software Analysis as a Service*. Such software evolution analyses are offered as services that can be accessed, composed into workflows, and executed over the Internet. This paper described a novel framework for composing such analysis services into workflows, consisting of a custom-made modeling language and a composition infrastructure for the service offerings. The framework exploits the RESTful nature of our analysis service architecture and comes with a service composer to enable semi-automated service compositions by a user. Our workflow language *SCoLa* takes advantage of the RESTful nature of our architecture. It comes with primitive activities such as service invocation and result query, and complex activities such as control flow (sequence, iteration, conditional flow) and parallel execution. Abstract workflows can be defined as templates for repeating service flows, as well as concrete workflows capture specific service compositions.

We validated our framework with an initial set of services which have been developed to fulfill some immediate analysis needs and are mostly based on analyses we previously developed for related projects, such as Evolizer [GFP09] and Change Distiller [FWPG07]. These services helped us populate the framework with enough analyses to provide varied, meaningful and non trivial evolutionary data for a first validation. As proof of concept, we presented two applications of *SCoLa* workflows using these services. Both cases showed the composition of many different types of analyses into a workflow, but with different purposes. The first application conceptually proves that our framework can be used to address relevant evolution analysis questions, such as finding code locations (i.e. hotspots) that have a high change frequency, intensive change coupling with other entities, and exhibit code clones. The second application shows how tools can harness such workflows to automatically gather a wide range of varied yet interlinked information about a software system and how they can use that for their own specific needs. In our case, we show how this data can be exploited to help stakeholders to gain a better understanding on a software, its history and quality from different perspectives, using intuitive visualizations.

The two applications presented originate from concrete evolution analysis needs we came across in our projects with industrial partners. However, they just show two possible uses of *SOFAS* and *SCoLa*. Several other tools can be built on top of them and many similar workflows can be defined according to the needs of an analyst. For future work, we foresee the definition of more ready-to-use workflow blueprints (abstract and concrete) to cover analysis scenarios reported both in the literature and gathered from industrial contexts by means of experiments.

Related approaches only allow the combination of analyses into predefined, unmodifiable sequences. Our approach enables users to compose and automatically execute them in a flexible way, based on the particular analysis needs. The composition we devised is only limited by the analyses offered, which is also one of its main weaknesses. In fact, to offer a wide range of potential workflow combinations, a substantial amount of diverse analyses is needed. With the current offering, only workflows working with data extracted from version control systems, issue trackers and plain source code can be created. In the future, we foresee the addition of several other services: from data gatherers to composite analyses targeting diverse evolutionary aspects or offering different algorithms (e.g. other change coupling or code clones detectors). We intend, however, to maintain the focus on software evolution. Thus, we do not plan to add software analyses that are not related to software evolution such as test coverage checks, performance analysis, or control flow analysis.

Nevertheless, the analyses currently registered in *SOFAS* are enough to fulfill concrete needs and to showcase the potential of our framework, in particular of its analysis composition features.

7

Replicating MSR Empirical Studies with *SOFAS*

Replicating MSR Empirical Studies with SOFAS
Giacomo Ghezzi and Harald C. Gall

Submitted for Journal Review.

Empirical studies play a vital role in software engineering. With them, researchers can rigorously prove or confute specific hypotheses usually originating from unreliable source of convincing knowledge such as common wisdom, intuition or speculation. The replication of such studies is just as fundamental and is one of the main threats to validity that empirical software engineering suffers. Such threats are manifold and range from lack of independent validation of the results, unavailability of the tools and methodologies used, to no impossibility to generalize the gained knowledge. This is particularly true for the Mining Software Repositories (MSR) field, in which it has been shown very few studies can be easily replicated. This paper shows how, with our *SOFAS* framework, we can alleviate such problem for this specific area of research. In particular, we can replicate, to different degrees of completeness, up to 60% of all the studies performing empirical studies published in the proceedings of the Working Conference on Mining Software Repositories. To further corroborate such claims, we replicate one of those studies and present the results.

7.1 Introduction

Empirical studies are a necessary yet difficult and time consuming means to better understand the software engineering phenomena and to help improve software development. Replication of such studies helps in further strengthening their findings and, in particular, in testing both their internal and external validity. With regards to external validity, replication helps defining the extent to which the results of an experiment can be generalized. That is, whether or not the experimental results depend on conditions specific to the original study. With regards to internal validity, replication helps researchers assessing the range of conditions under which the results stand [SCVJ08].

Mining software repositories (MSR) has become in the years a fundamental area of research of software engineering in that it helps to better understand and support software development. This area has a strong foundation on empirical studies, however, it is still missing a systematic approach to replicability. Thus, as reported by Robles [Rob10], very few published studies can be easily replicated. The problem lies in both the tools used and the data used by such tools.

Tools are available for only around 20% of the studies and another 20% are only partially available. Moreover, even when publicly available, they are difficult to set up and use. As a matter of fact, they are mostly prototypes (or just a collection of scripts) working only under specific operating systems and settings and are usually offered "as-is" without any user manuals or technical support.

Data can be divided into "raw" data and processed data. The first one can be directly retrieved from publicly available sources such as version control systems, issue trackers, plain source code, mailing lists, etc. The second one, which is what is actually used by researchers to perform their analyses, is the result of the retrieval and processing of such raw data. While "raw" data is usually widely available (at least in the case of OSS projects), processed data is not.

Different approaches have already been proposed in the community to try to address this problem. These efforts are mainly aimed at creating large, internet accessible, software analysis data repositories. Some of them offer a queryable, static collection of data for specific projects fetched from single [Moc09] or multiple sources [NZ10, SNL+06], while others allow the user to interactively run specific analyses on its own projects of interest [Gob08, GS09] through online web applications. Large, static software data repositories give third party applications and analyses a sizable, common body of knowledge to build upon. They can be also be extremely useful to provide benchmark data to test

and compare similar tools/analysis that use such data. However, they do not target the replication of the actual analyses and are based on a fixed list of software projects. The more interactive approaches solve this issue, however they still limit the user to only the analyses they implement. Replicability is thus still limited to very specific cases. While this is a welcomed step in the right direction, a broader and more systematic approach to replicability is still missing. We claim that a solution like our *SOFAS* (SOFtware Analysis Services) [GG11] platform can fulfill such need.

SOFAS is a RESTful software analysis architecture which was originally created to support a lightweight, flexible interoperability of distributed analyses. It is made up of three main constituents: Software Analysis Web Services, Software Analysis Ontologies and a Software Analysis Broker. Software Analysis Web Services offer different software evolution analyses as standard RESTful web service interfaces. They adhere to specific Software Analysis Ontologies defining and representing the data they consume and produce. The Software Analysis Broker acts as the services manager and the interface between the services and the users. It contains a *Services Catalog* of all the registered analysis services with respect to a specific software analysis taxonomy. Such analyses are accessible via a single entry point and easily invokable by information such as the URLs of the source control repository, the issue tracking system, or release notes, etc. Moreover, through the Software Analysis Broker, users–using a custom web UI–and machines–using dedicated REST endpoints–can combine services into analysis workflows to perform specific, composite, analysis tasks.

In our opinion, *SOFAS*' very core features are key in facilitating and supporting the replication of software evolution analyses. In fact, having analyses as RESTful web services with a uniform interface improves accessibility. Users don't have to install or configure any tool, but just need to supply the service with the necessary data and fetch the results upon completion. These results are then easily available to all the other *SOFAS* users as they can be retrieved online, straight from the analysis itself. The use of public, well defined semantic web ontologies to describe analysis results facilitates the interpretation of such data. Not only they describe in clear way the domain of discourse, both semantically and syntactically, but they also come with a powerful, standard query language, SPARQL [PS08].

To evaluate the replicability potential of *SOFAS*, we perform a complete literature review off all the papers published at MSR, select all the ones based on empirical studies and determine whether they can be replicated or not with it. We show that we can replicate to different degrees of completeness up to 62% of these studies. Studies that can be fully

replicated account for 30% of the total, while the remaining 32% are studies that can only be partially replicated. This means that we can produce all the ground data needed, but we miss the analysis that extracts the results needed in the study from it.

The remainder of the paper is structured as follows: in Section 7.2 we give a brief overview of *SOFAS* (for a detailed description of the architectural aspects we refer to [GG11]). Then, in Section 7.3, we present the method used in this study to assess the replicability support offered by *SOFAS* and show its results. In Section 7.4, we present in detail a replication use case in which we replicate one of the empirical studies found in the previous section, to better show the potential of *SOFAS* in such a replication context. Section 7.5 gives an overview of the related work.We then conclude with a discussion of the strength and weaknesses of the approach.

7.2 *SOFAS*

SOFAS is a RESTful architecture devised to provide software analyses as RESTful web service and to combine them into analysis workflows in a easy and effective way. Its architecture, shown in Figure 7.1, is made up by three main constituents: Software Analysis Web Services, a Software Analysis Broker, and Software Analysis Ontologies. Software Analysis Web Services expose the functionality and data of software (evolution) analyses through a standard RESTful web service interface. The Software Analysis Broker, acts as the services manager and the interface between the services and the users. It contains a catalog of all the registered analysis services. Moreover, it also offers interfaces for both humans and machines to compose such services into workflows and execute them. Ontologies define and represent the data consumed and produced by the different services. In the following we briefly describe each of these three components.

7.2.1 Software Analysis Web Services

From a user's perspective, software analyses are inherently linear and uniform in the way they work. Given some information about a software project (be it the code, its source code repository, some data already calculated by an analysis, etc.) and possible analysis calibration settings, they extract and/or calculate their specific data. Once that is completed, the results can be fetched. Such linearity and uniformity make REST a perfect fit for our needs, as some of its inherent principles are also the main requirements and characteristics

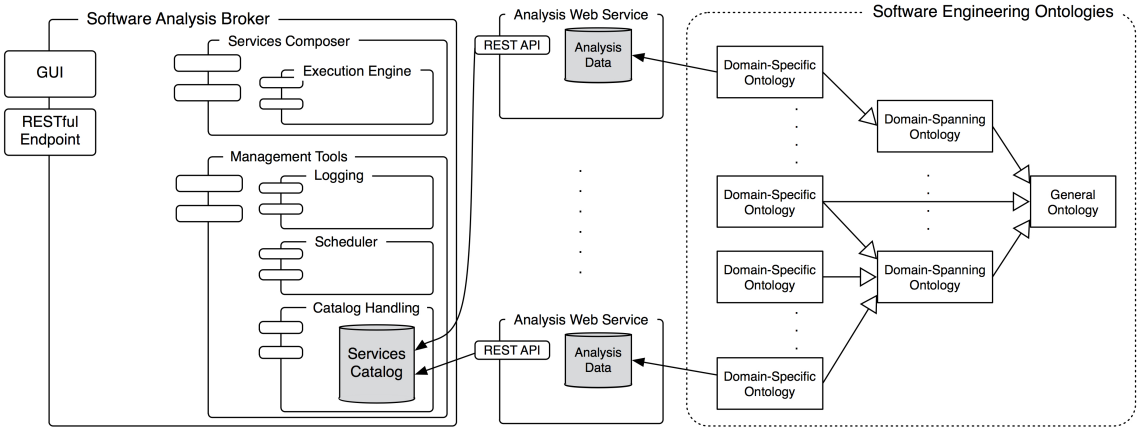

Figure 7.1: *SOFAS* overall architecture

of our services.

A RESTful web service provides a uniform interface to the clients, no matter what it actually does. It is a collection of resources all identified by URIs, which can be accessed and manipulated with HTTP methods (e.g., POST, GET, PUT or DELETE). Furthermore, every message exchanged is self-descriptive as it always contains the Internet media type of the content, which is enough to describe how to process it.

These analyses can be roughly divided into three categories: **Data gatherers**, **basic** and **composite software evolution analyses**. **Data gatherers** work on raw data, extracting it from different software repositories, such as version control, issue tracking, mailing lists, plain source code, etc., and importing such vital ground data into *SOFAS* for other analyses to use it. **Basic software evolution analyses** exploit the data imported by one of these data gatherers to calculate all sort of software evolution information. This can be version history metrics, code metrics of specific releases/revisions, issue tracking metrics, etc. **Composite software evolution analyses** aggregate data produced by other analyses to calculate more complex and domain spanning evolution information. For example, the bug proneness of files given the issue tracking and version histories of a software project. Another example is the detection of code disharmonies [LM05] in a software, given the its source code metrics.

7.2.2 Software Analysis Ontologies

To describe the data produced by those analyses, we have developed our own family of ontologies, called *SEON* (Software Engineering ONtologies)[1]. An ontology is a formal description of the important concepts identified in the domain of discourse and their relationship to one another [Gru93]. It provides a common vocabulary for a specific domain, which can be used to express the meta-data needed to capture the knowledge of the exchanged, shared, or reused data. Our ontologies, defined in OWL, are organized in a pyramidal structure. The top layer comprises ontologies describing general concepts, the attributes to describe them, and the relations between the concepts. The second-highest layer defines domain-spanning concepts. These concepts describe knowledge that spans a limited number of subdomains. The third layer is made up of ontologies describing different domains corresponding to important aspects of software evolution, e.g. issue and version management. At the bottom of the pyramid sit ontologies describing system-specific or language-dependent concepts (*e.g.*, Java-specific constructs, SVN concepts, etc.). We refer to [WGH+12] and *SEON*'s official web page for more details on the approach and for a complete description of these ontologies.

7.2.3 Software Analysis Broker

The Software Analysis Broker offers a single entry point to *SOFAS*' services. Through it, they are kept track of, classified in a registry, queried, monitored and coordinated. In this way, users do not have to interact directly with the raw services. As shown in Figure 7.1, this component is in turn made up of four main sub-components: the *Services Catalog*, a user interface, the *Services Composer*, and a series of management tools.

The *Services Catalog* stores and classifies all the registered analysis services so that a user can automatically discover services, invoke them, and fetch the results. We developed a software analysis taxonomy to systematically classify existing and future services. This taxonomy divides the possible analyses into three main categories: development process, underlying models, and source code. For more details we refer to the *SOFAS* website[2].

The User Interface is the access point to the Software Analysis Broker. It consists of a web UI, meant for human users and a series of RESTful service endpoints to be (semi)-automatically used by applications. Through the UI the user can easily browse

[1] www.se-on.org

[2] https://seal.ifi.uzh.ch/sofas

through the *Services Catalog* to check for analyses offered, select the ones of needed and, if necessary, combine them into workflows. The user interface offers an intuitive, high level way to do that, allowing the user to combine the services in a "pipe and filter" fashion. The user can also pick from some already predefined combinations of analysis services provided as high level analyses workflows.

The *Services Composer* takes care of translating the workflows defined through the UI into actual, executable ones and execute them. Having the composition definition and the actual composition language decoupled, allows the user to compose services in a intuitive way, hiding the complexity and technicalities of the actual composition and orchestration. Moreover, calls to additional management services can be automatically weaved into a user defined workflow.

Management services are used to support a correct execution of analysis workflows. In fact, such workflows are usually made up of long running, asynchronous web services. Thus, in order to effectively handle them, every single service needs to be logged and monitored to check if is up and running, if it is in an erroneous state and why, if it completed a required operation, etc.

7.2.4 *SOFAS* and Replication

A successful study replication can occur only if the following three aspects are fulfilled:

- **Availability of ground data.** The data on which the study is based should be easily and readily accessible in some form, preferably over the Internet.
- **Availability of the analysis itself.** The tools or scripts used to perform the study– which handle and analyze the ground data to produce the final results–should be publicly available and usable. If not, detailed instructions on how to perform the analysis, or even the algorithms, should be provided.
- **Availability and traceability of results.** The results produced in the study should be available in the same way as the ground data. This is not strictly vital for the actual replication. However, it facilitates the verification of the results and claims of the original study and the comparison with the results of the replicas.

Wuersch et al. [MWG10] already made a case for using the semantic web to obviate the issue of availability and traceability of the results in mining software repositories analyses. In our opinion, the combination of semantic web and REST, which is the core foundation

of *SOFAS*, is a perfect fit in fulfilling the three aforementioned aspects and thus facilitating and supporting replicability.

Having analyses as RESTful web services with a uniform interface improves their accessibility. Users do not have to install or configure any tool, but just need to supply the analysis service with the necessary data. Moreover, services can also be easily integrated into custom user application and scripts. In fact, being RESTful, they can be called with simple HTTP methods, without the need of custom libraries or frameworks. For example, this could come down to a single command issued to a simple command-line tool like *cURL*. With this solution, also the results are available online to all the other *SOFAS* users as they can be retrieved online, straight from the analysis itself. For example, given one of the services extracting the complete history of a version control repository reachable at `http://habanero.ifi.uzh.ch/gitImporter`, the results of specific analyses are available at `http://habanero.ifi.uzh.ch/gitImporter/analyses/<analysisname>`. Depending on the user's need, these results can be browsed online, fully downloaded in their RDF form or queried online using SPARQL. In this last case, the user has to encode the query in the URL, e.g., `http://habanero.ifi.uzh.ch/gitImporter/analyses/<analysisname>?query=<sparqlquery>`, or use the service web UI that has a form through which queries can be composed and ran.

The use of public, well defined semantic web ontologies to describe these analysis results facilitates their interpretation. In fact, not only they describe in a clear way the domain of discourse, both semantically and syntactically, but they also come with a powerful, standard query language, SPARQL [PS08]. This means that researchers can also make the queries they used public to be re-used or verified straight away. Moreover, the semantic web concept of statements represented by triples of URIs enables us to link and query data that is stored on different services and build an internet-scale graph of analysis information. In fact, all *SOFAS* services, in addition to a standard SPARQL endpoint to query the analysis data, also expose a URI for every piece of information they extracted so that it can be dereferenced over the Internet. For example, let's consider one of *SOFAS*' issue-revision linker services which, given the issue tracking and version histories of a specific software project (extracted by other services), reconstruct the links between issues and the revisions that fixed them. Such service would produce triples like this:

```
habanero.ifi.uzh.ch/bugzillaImporter/analyses/test/issues/124
se-on.org/ontologies/domain-spanning/integration-history-issues/isFixedIn
habanero.ifi.uzh.ch/gitImporter/analyses/test/changesets/1239
```

From such triples, client applications, as well as humans, can easily follow the links to access the actual resources involved or use them as an input for further SPARQL queries. From a very small initial piece of information, users can incrementally navigate the information space and expand their knowledge about the system being used. Figure 7.2 gives an example of such information space and how it is distributed between different services and can be navigated. For the sake of simplicity we just show one-directional links, however, the information graph can also be traversed in the other direction with the use of opposite properties. At last, ontologies were explicitly designed to be shared and can thus be serialized using the RDF/XML standard and exchanged without the loss of data semantics.

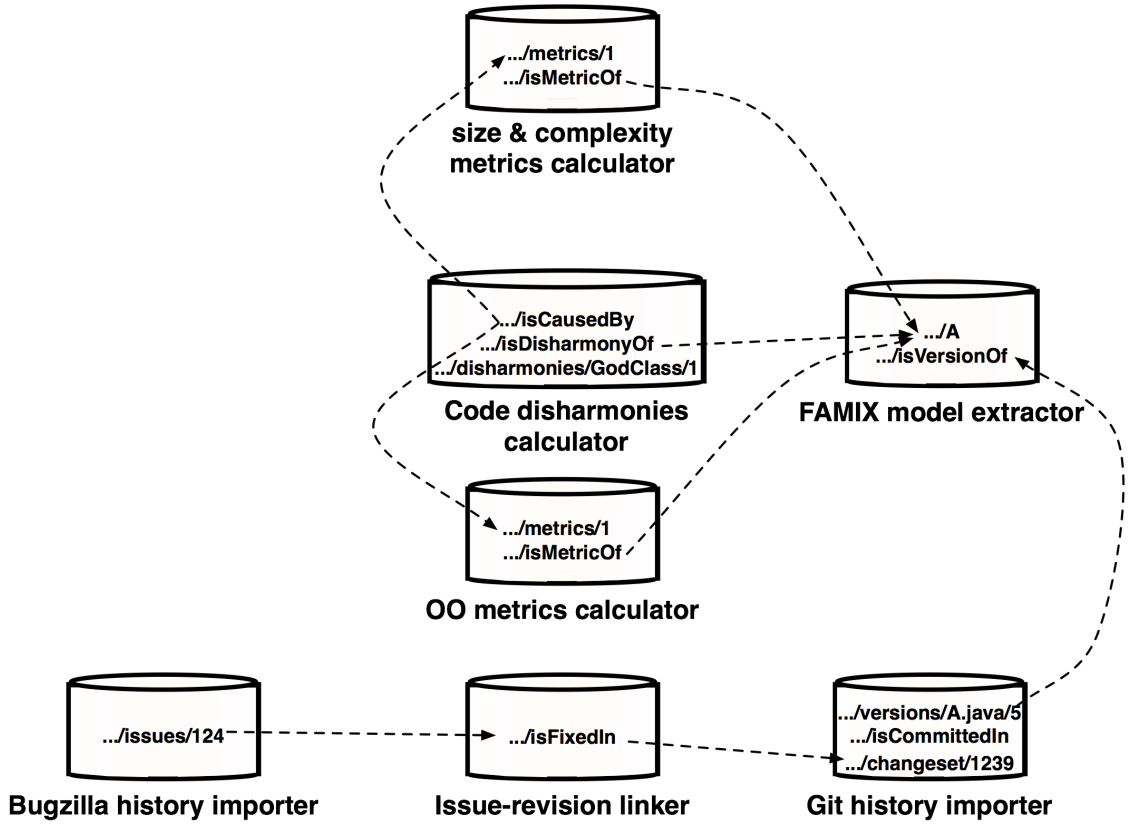

Figure 7.2: An example of the distributed graph of analysis data produced by *SOFAS*.

SOFAS has an entire sub-family of analysis services called **data gatherers**, which we already introduced earlier and which sole purpose is to import raw, ground data in *SOFAS* to be used by other services, as well as human users. In this way, the ground data can be

easily extracted, accessed and used as we just described for the other analysis services.

Finally, the ability to compose services into workflows allows users to execute more complex analyses building on the existing offering. This not only broadens the amount of existing studies and analyses that can be replicated, but can also be exploited in creating and executing novel ones.

7.3 Experimental Evaluation

To show the applicability of our framework in replicating MSR studies, we performed a complete literature review of all the works published in the proceedings of all MSR conferences up to 2011 (a total of 8 editions). We then filtered out all the papers that did not present empirical studies and which could thus not be replicated. Such papers, which actually account for the 51% (88 papers) of the total, usually propose new methods, analysis tools and frameworks, visualizations, case studies, etc. The remaining papers were then classified into 6 broad categories. These categories were subjectively constructed and were kept rather generic on purpose. In fact our goal was to identify the main "macro" empirical research areas and check whether our approach was particularly successful on any of those. We did not intend to perform a through and detailed review of the field.

9% of the 84 empirical papers we found, deal with the plain *mining of version history* data from version control systems for different purposes; to shed light on the development process [MAH10], to better understand how developers work [LSWG04] and their dynamics [WS08]. The same number of papers address the issue of *bug prediction*, i.e., finding code entities that are most likely to be fixed or to be buggy based on different historical information, . The work by Giger et al. [GPG11a] and by Sliwerski et al. [SZZ05b] are two prime examples of such category. The two largest categories, both accounting for 22% of the total, are *defect analysis* and *social networks and team analysis*. The first one deals with the analysis of all sort of defects, e.g. clones [RBD10], reported bugs [AM07], etc. and the exploitation of such data. The second one deals with the extraction and analysis of the social networks associated to a software project [BGD+06] and the team/development dynamics of it [MM07]. 20% of the papers belong to a more generic *historical mining* category, which encompasses studies exploiting all sort of software development-related historical data for a wide range of applications. For example, deducing a developer's expertise based on its source code contributions to ease bug assignment [MKN09] or finding license violations [HKVD11]. At last, 15% of the papers deal with *change anal-*

ysis. That is, the analysis of the changes performed on the source code to uncover or extract more information about how the software changed. For example, studying how identifiers are renamed [EAP+11] or extracting and analyzing commonly occurring change patterns [KWB05].

The empirical study of every selected paper was then inspected in detail to assess if and how it could be replicated using *SOFAS*. As a result, the studies were then rated as **fully supported**, **partially supported** and **not supported**. A study was considered **fully supported** only if the same results could be replicated or if all the necessary data could be calculated with the exception of its final aggregation or interpretation. On the other hand, a study was considered **partially supported** if its results could not be replicated out of the box, but the ground data from which they are derived could be calculated. At last, a study was deemed **not supported** if no, or very little, ground data could be calculated.

Replication is usually divided in two main categories: exact and conceptual. Exact replication is when the procedures of the experiment are followed as closely as possible. Conceptual replication is when the experimental procedure is not followed strictly, but the same research questions or hypotheses are evaluated, e.g. different tools/algorithms are used or some of the variables are changed [SCVJ08]. In this paper, we did not distinguish between exact and conceptual replication. A study was considered replicable whenever it could be replicated, either conceptually or exactly replicate, using a service or a combination of services currently available in *SOFAS*. Appendix B illustrates how all the fully supported studies can be concretely replicated with *SOFAS*.

The results of such replicability assessment are reported in Table 7.1. The level of replicability is spread quite evenly across the different research categories and there is no category for which *SOFAS* was more successful than the rest. The only exceptional category is *Historical Mining*, as no full replicability could be achieved for it. This is mostly likely due to the fact that it is a quite broad category encompassing very diverse studies each needing their own very specific analyses to calculate the final results needed. If such analyses are not present in *SOFAS*, the studies can not be replicated. Nevertheless, 49% of such studies are partially supported, this is because a lot of them are based on data extracted from software repositories such as version control, issue tracking, mailing lists and plain source code, which are well covered by *SOFAS*. The other categories are more specific and deal with more common issues in software evolution analysis. That means that analyses dealing with such issues are more likely to already be existing in *SOFAS*. Moreover, studies in these categories are often similar to each other and can thus be conceptually replicated using the same analyses.

Research category	**Number of papers**	**Fully replicable papers**	**Partially replicable papers**	**Non replicable papers**
Version History Mining	8 (9%)	4	0	4
History Mining	17 (20%)	0	8	9
Change Analysis	13 (15%)	5	6	2
Social Networks and People	19 (22%)	6	5	8
Defect Analysis	19 (22%)	8	6	5
Bug Prediction	8 (9%)	2	2	4
	88 (100%)	25 (30%)	27 (32%)	32 (38%)

Table 7.1: The results of the replicability evaluation.

In the following section, we replicate an empirical study found with this assessment to better show and prove how *SOFAS* concretely supports such replication.

7.4 A Replication Use Case

In this section, we present in details the replication of a study published at MSR that we replicted with *SOFAS*. In particular, we (1) introduce the original study, (2) describe what we intend to replicate, (3) show how that is concretely done with *SOFAS*, and (4) present and discuss the results of the replication.

7.4.1 Do time of day and developer experience affect commit bugginess?

This study, performed by Eyolfson et al. [ETL11], investigates the correlation between the bugginess of a commit and a series of factors: the time of day of the commit, the day of week of the commit, the experience and commit frequency of the committer. Such analysis is based solely on the history of a project extracted from its version control system. The

authors consider a bug-introducing commit any commit for which there exists another commit explicitly fixing it later in time. To find them, they first detect all the bug fixing commits using a standard heuristic used in the field. That is, finding the ones that have specific keywords (e.g. “fix”, “fixed”, etc.) in their commit message. Buggy commits are then the ones that last changed files that were involved in such fixes.

In their investigation, the authors studied the version control histories of the Linux kernel and PostgreSQL and uncovered four main findings. First, about a quarter of the commits in a project history introduce bugs. Second, the time of the day influences the introduction of bugs, as late night commits (submitted between midnight and 4 AM) are significantly buggier and commits between 7 AM and noon are less buggy. Third, regularly committing developers (daily-committers) and more experienced committers introduce less bugs. At last, the influence of the day of the week on the commits bugginess is project-dependent.

In this paper, we prove these four findings by fully replicating the original study. Moreover, we also test if such findings hold for three other popular OSS projects: Apache HTTP, Subversion and VLC. The results of this replication are available online from the services used in the study. This data can be accessed in restricted read mode with the following guest account, username: REPLGUEST, passwordREPLGUEST2012.

Replication Set-up

To replicate this study, we take the following steps:

1. **Extract the complete project version control history.** This is accomplished using one of the version control history extractors currently registered in *SOFAS*. As of now, six of such services targeting different version control systems exist. They handle git (two of them), cvs, mercurial and svn (two of them) repositories. In this case, we used one of our git services. The results are available at:

   - `habanero.ifi.uzh.ch/newGitImporter/httpd`
   - `habanero.ifi.uzh.ch/newGitImporter/postgresql`
   - `habanero.ifi.uzh.ch/newGitImporter/linux`
   - `habanero.ifi.uzh.ch/newGitImporter/subversion`
   - `habanero.ifi.uzh.ch/newGitImporter/vlc`

2. **Find all the bug-introducing and bug-fixing commits (a.k.a. revisions) from such history.** This is also accomplished by one of the currently registered bug-revision linkers. These services extract such information from a project history extracted by one of the aforementioned version control services. Currently, five of them exist. Three of them are version control system independent but only find the bug-fixing commits. This is because this is the only information that can be recovered directly from version control history alone. Finding bug-introducing commits is, on the other hand, more complex to recover and the process to do so is version control system-dependent. The two linkers currently supporting this feature target git and mercurial-based repositories. These services are based on algorithms that are very similar to the ones used in the original study. The main difference lies in the heuristics used to detect bug-fixing commits. In fact, they consider a commit a bug fix only if it contains the term *fix*, while our heuristics are based on a larger vocabulary (e.g. *fixes*, *fixed*, *bug(s)*, etc.)

3. **Extract the commit frequency and experience of the all the users who introduced bugs (calculated from the bug introduction date).** This is achieved by querying the data extracted in the first step with specific SPARQL queries.

4. **Aggregate the buggy commits by time of the day, day of the week, developers experience and commit frequency.** This is also achieved with SPARQL queries (for the actual queries used, we refer to Appendix A).

5. **Final results interpretation.** *SOFAS* simply supports the extraction and combination of analyses and data. The conclusions have still to be manually drawn by the users of such analyses, depending on their specific needs. The links found are available at:

   - `habanero.ifi.uzh.ch/bugFixesLinker/httpd`
   - `habanero.ifi.uzh.ch/bugFixesLinker/postgresql`
   - `habanero.ifi.uzh.ch/bugFixesLinker/linux`
   - `habanero.ifi.uzh.ch/bugFixesLinker/subversion`
   - `habanero.ifi.uzh.ch/bugFixesLinker/vlc`

Results

We list here the results of this replication divided into the original four main findings. All the projects were analyzed between July 1st and July 10th, 2012.

Percentage of buggy commits

Our replication confirms the results of the original study for both Linux and PostgreSQL. The slightly different values can be explained by the different heuristic used to detect bug fixes and the different analysis date (the projects were analyzed a year later than the original study). Moreover, all the other analyzed projects exhibit similar values (22-28%), as shown in Table 7.2. These results seem to indicate a trend worth investigating more in detail, with a larger body of projects.

	# commits	# bug-introducing commits	# bug-fixing commits
Linux	268820	68010 (25%)	68450
PostgreSQL	38978	9354 (24%)	8410
Apache Http Server	30701	8596 (28%)	7802
Subversion	47724	12408 (26%)	10605
VLC	47355	10418 (22%)	10608

Table 7.2: Small summary of the characteristics of the analyzed projects

Influence of time of the day on commit bugginess

Figures 7.3 to 7.7 show the correlation of the time of the day of a commit with its bugginess. The graphs compare the time of the day of each commit on a 24-hour clock (in the committer's local time) to the percentage of bug-introducing commits. The horizontal line in the graphs indicates the overall percentage of buggy commits for each project. Our replication confirmed the results of the original study for both original projects. Moreover, the analysis of the additional projects corroborates the finding that the amount of commits introducing a bug is particularly high between midnight and 4 AM and that then it tends to drop below average in the morning and/or early afternoon. However, these 'windows' of below average bugginess greatly vary between projects. Furthermore, the projects'

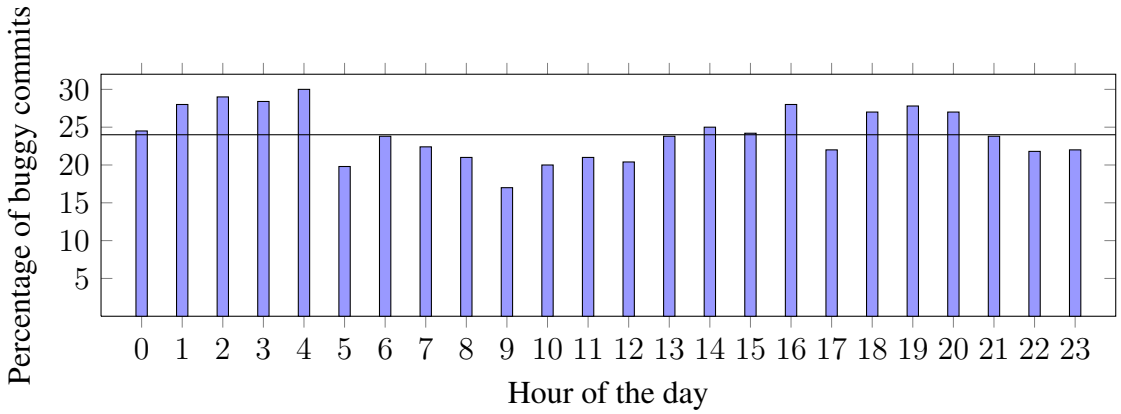

Figure 7.3: Percentage of buggy commits versus time-of-day for PostgreSQL

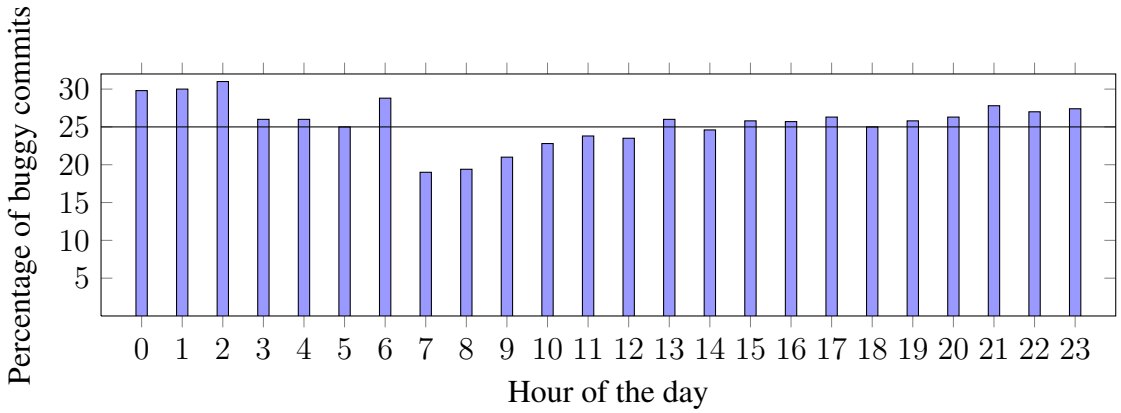

Figure 7.4: Percentage of buggy commits versus time-of-day for Linux

bug commit bugginess follows very different patterns which do not allow any further generalization on the influence of the time of the day on the commit bugginess.

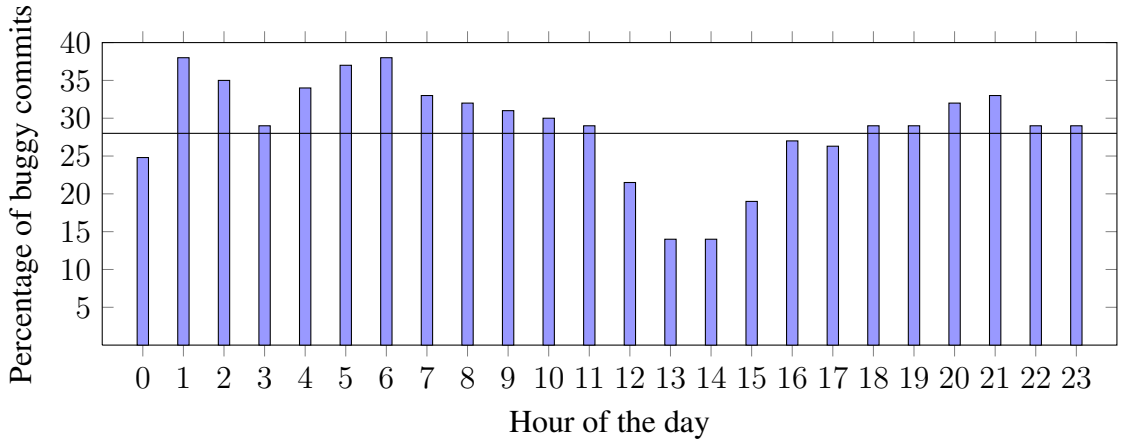

Figure 7.5: Percentage of buggy commits versus time-of-day for Apache HTTP Server

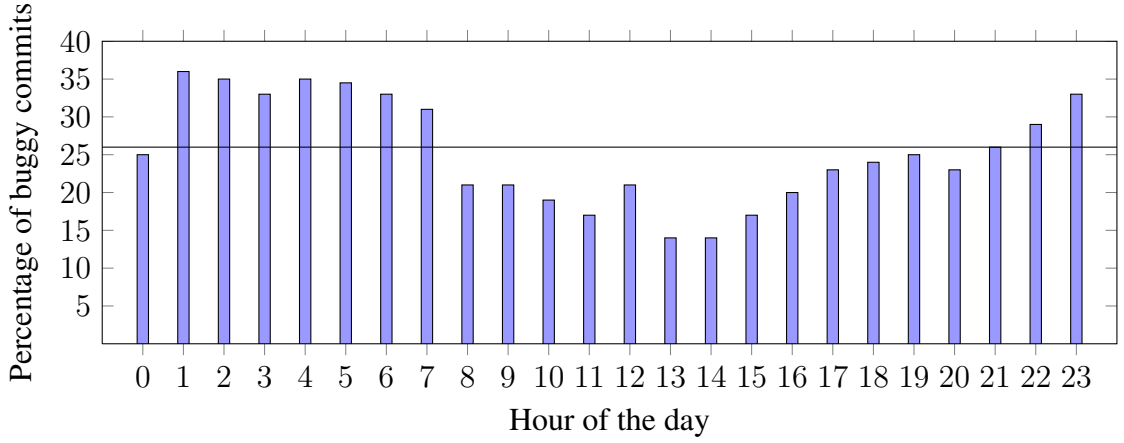

Figure 7.6: Percentage of buggy commits versus time-of-day for Subversion

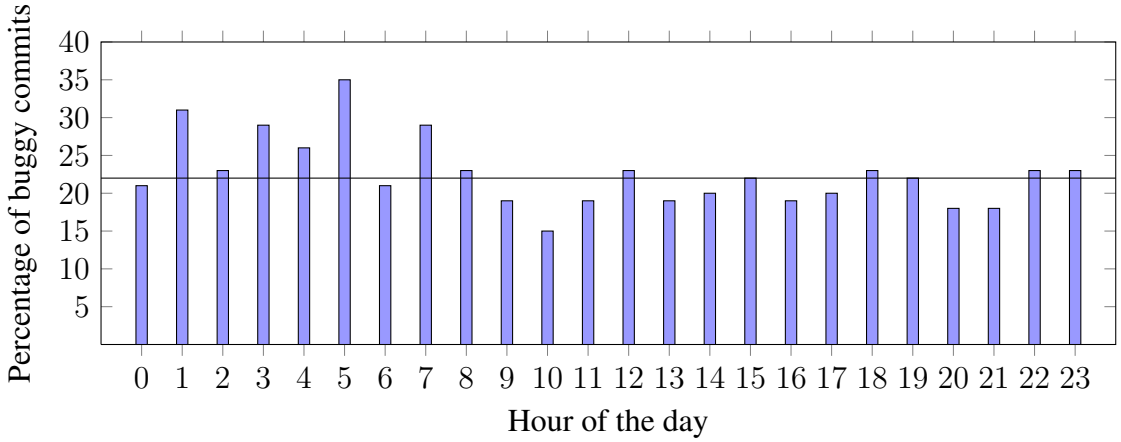

Figure 7.7: Percentage of buggy commits versus time-of-day for VLC

Influence of developer on commit bugginess

Figures 7.8 to 7.12 correlate author experience at time of commit to the bugginess of the commit. Our replication confirms the original results that bugginess decreases with increased author experience for all the projects analyzed. In all projects, a drop in commit bugginess is evident as the time a developer spent of a project increases. In four of the projects such drops happens between 32 and 40 months of experience, while for the remaining one, PostgreSQL, such drop takes place much later, at 104 months of experience.

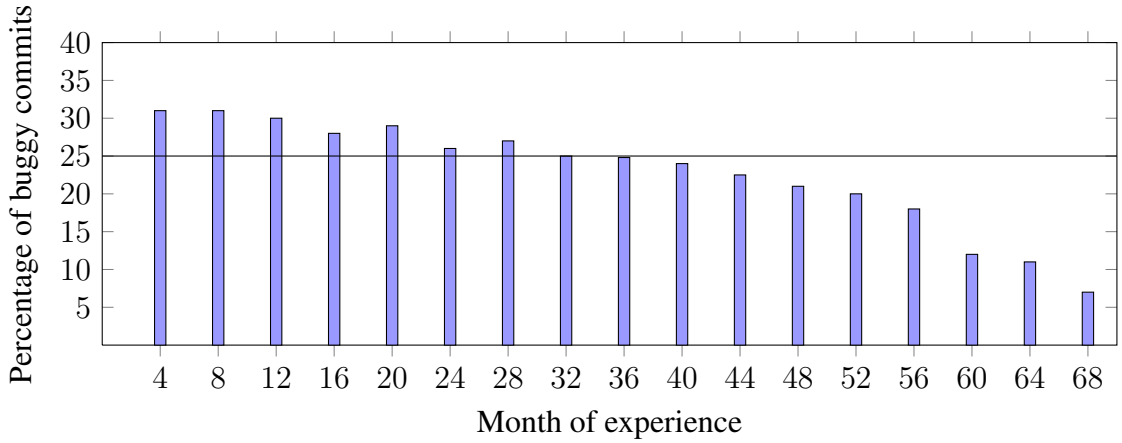

Figure 7.8: Percentage of buggy commits versus author experience for Linux

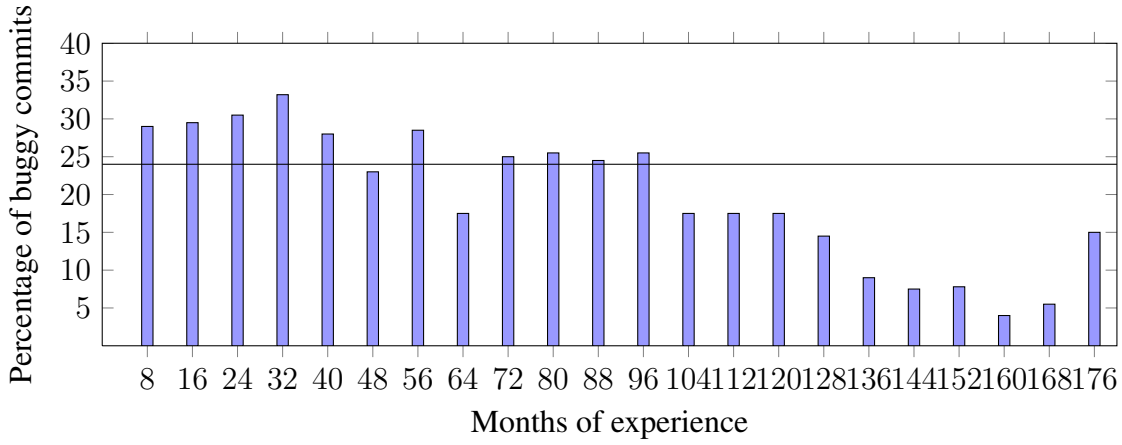

Figure 7.9: Percentage of buggy commits versus author experience for PostgreSQL

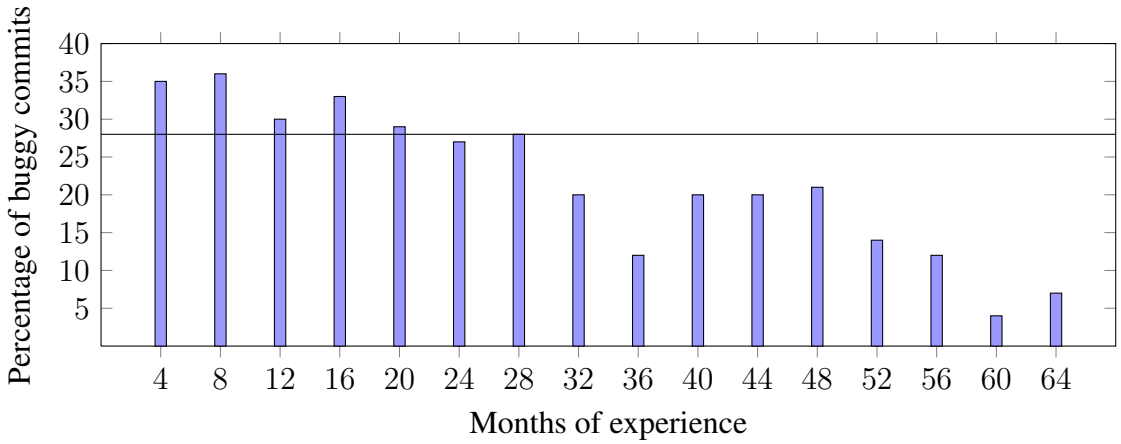

Figure 7.10: Percentage of buggy commits versus author experience for Apache HTTP server

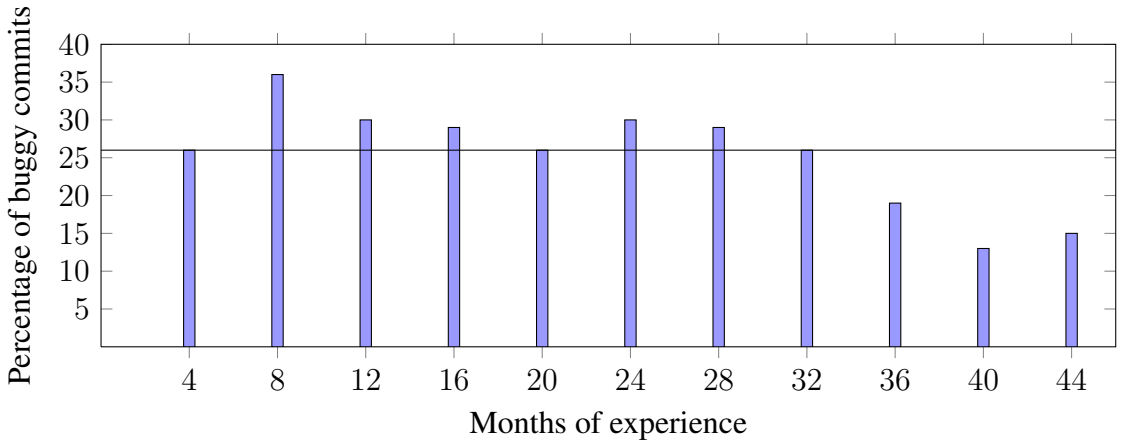

Figure 7.11: Percentage of buggy commits versus author experience for Subversion

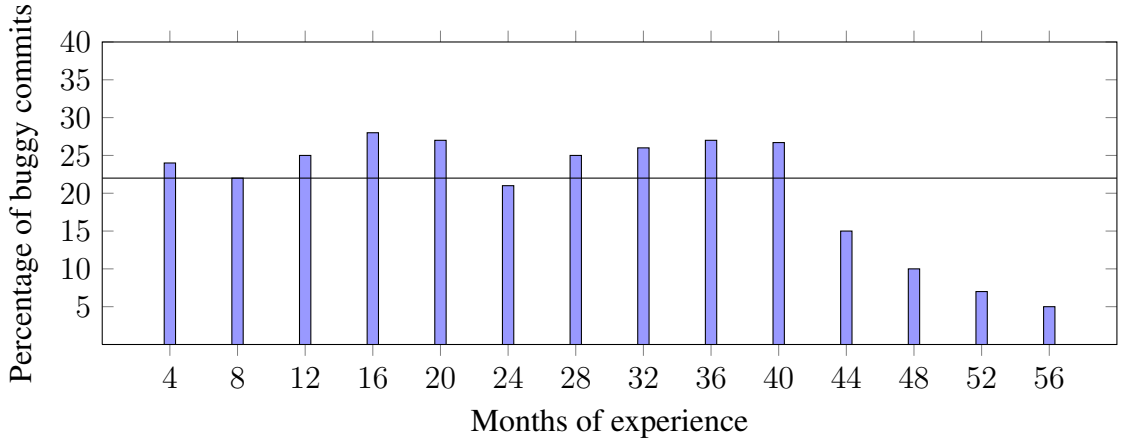

Figure 7.12: Percentage of buggy commits versus author experience for VLC

Influence of day of the week on commit bugginess

Figures 7.13 to 7.17 show the correlation between the day of the week with the commit bugginess on that day. As before, the solid horizontal line represents the overall commit bugginess of the project. Our results confirm the results of the original study. However, the additional projects' bugginess present very different patterns. Apache HTTP server and Subversion tend to have two commit bugginess 'phases': a higher than average one from Tuesday to Friday and a lower than average one from Saturday to Monday. On the other hand, the bug introduction in VLC is almost the opposite, as it lower in the middle of the week (Wednesday to Friday). The analysis of these additional projects shows that the finding of the original project that commits on different days of week have about the same bugginess is not generalizable. Moreover it also shows that the results of a previous similar study [SZZ05b] that showed the Friday was the day with the most buggy commits (based on the analysis of Mozilla and Eclipse) cannot be generalized.

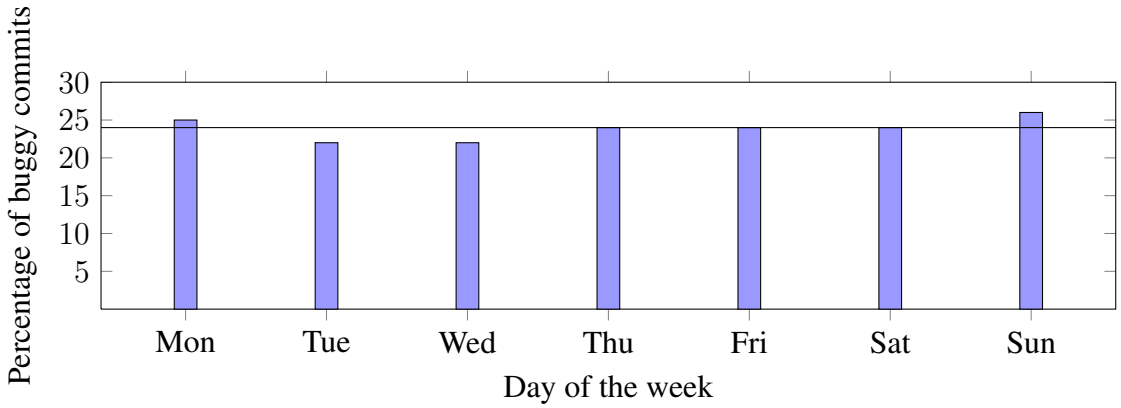

Figure 7.13: Percentage of buggy commits versus day-of-week for PostgreSQL

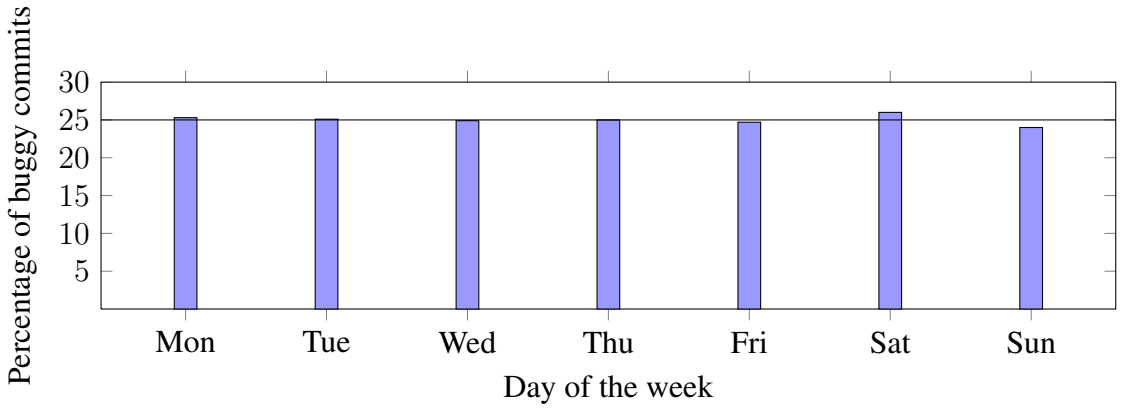

Figure 7.14: Percentage of buggy commits versus day-of-week for Linux

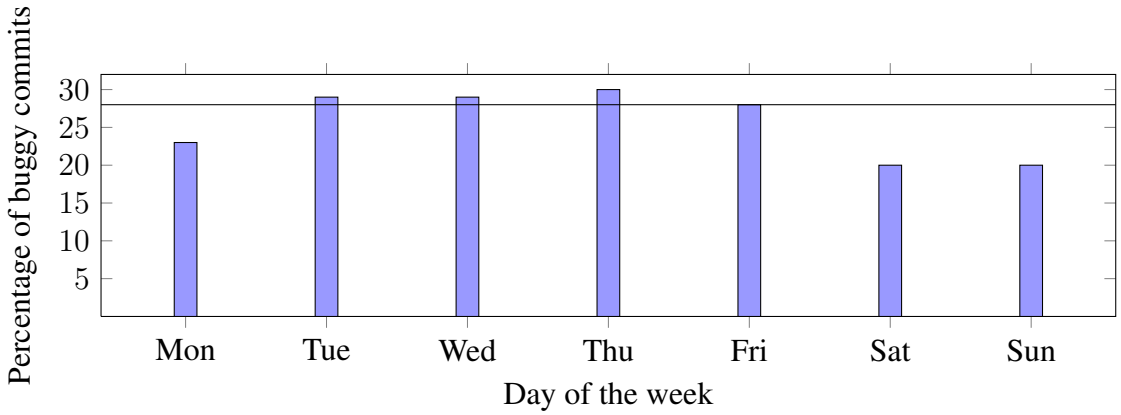

Figure 7.15: Percentage of buggy commits versus day-of-week for all Apache HTTP server

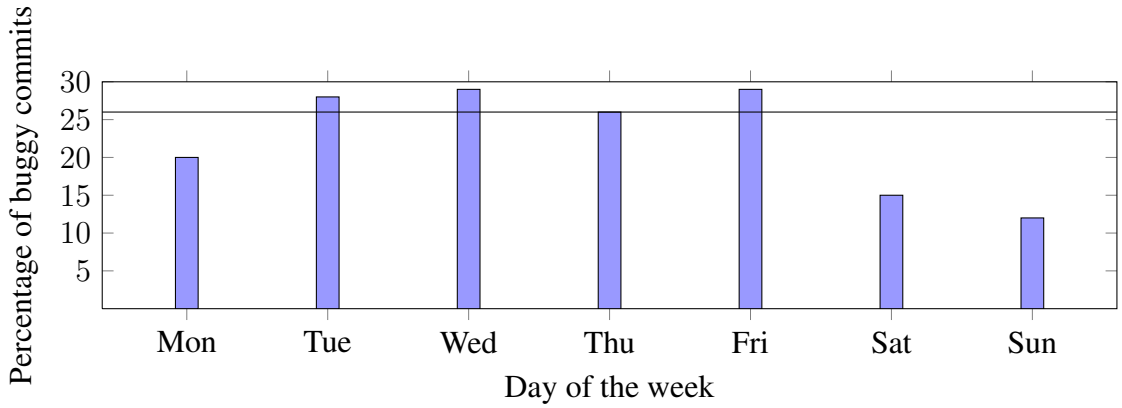

Figure 7.16: Percentage of buggy commits versus day-of-week for Subversion

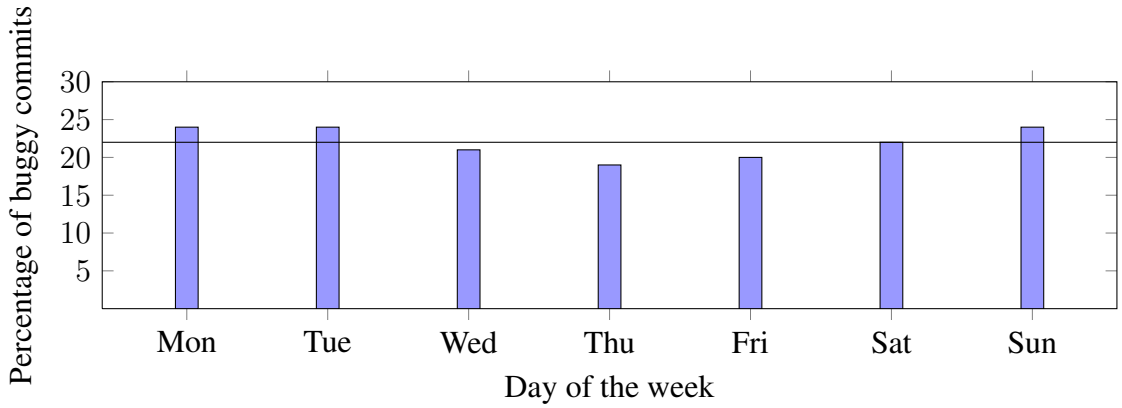

Figure 7.17: Percentage of buggy commits versus day-of-week for VLC

7.5 Related Work

The replication of studies is an essential task to expand and mature the current body of knowledge of any branch of science or technology. In particular, it is vital in gaining a deeper understanding on which results or observations hold under which conditions [BSL99]. However, such task is also intrinsically difficult, primarily because it is hard

to reproduce a setting that is the same–or extremely similar–as the original study. Because of that, software engineering is still short on systematic and widespread replication. These issues have already been acknowledged and widely investigated especially in the empirical software engineering community [BSL99, SCVJ08, Mil00]. This has led, for example, to the introduction and successive refinement of replication packages [MWM97, KL95] (also known as lab packages [BRW+08]) and data repositories. A replication package is a detailed description of an experiment, containing all the material needed to run the experiment (data collection forms, definition and planning of the experiment, how to analyze the collected data, etc.). Data repositories, host publicly available software engineering data sets containing all sort of data: error data, failure data, software metrics, software cost, statistical methods, analysis tools, etc. Such repositories are mostly used as commonly accepted benchmarks to validate models and studies. One of the most prominent example is PROMISE [GBO07].

In the subfield of software evolution analysis/mining software repositories, however, such systematic approaches are still lacking. This was brought to light by Robles' survey [Rob10] on the potential replicability of the empirical studies presented at the Working Conference on Mining Software Repositories (MSR) throughout the years. That work shows that a systematic approach to replication is still lacking and that very few studies can be effectively replicated. The main reason being the unavailability of the tools, scripts and instructions necessary to run the study (80% of the cases) and the actual results data set (only six papers). Taking inspiration from this paper, we wanted to prove that our *SOFAS* framework and its underlying architectural rationale are a viable solution to ease such unsolved issues.

Similarly to the PROMISE repository approach, researchers have established online software evolution data repositories. Flossmole [HCC06] is probably the biggest and most notable one, containing nearly 1 TB of data extracted from all the major code source forges (sourceforge, github, freshmeat, etc.), covering the 2004-now period. This includes the metadata of more than 500,000 open source projects, automatically scraped from their forge web page. Such metadata includes: the programming language(s) used, the platform(s) supported, the license, the number of developers and brief information about them (name, username, email), developer's role on the project, issue-tracking data (e.g. date opened, status, date closed, and priority), etc. All this data focuses on more high-level development information and dynamics and is offered "as is". No actual analysis is performed, nor the project source code is investigated. The Ultimate Debian Database [NZ10] follows a similar approach but only for the Debian Linux distribution

and all its binary packages. However, it focuses on extracting and presenting more system specific information (package popularity, history of packages upload, etc.) to try to countermeasure the lack of a proper development infrastructure that other Linux distribution (e.g. Red Hat and Ubuntu) suffer from. Mockus [Moc09], on the other hand, collected and indexed the version control history and the actual source code of a large sample of software projects from the most notable forges. The author discusses the methods developed to build such dataset, but the actual data is not publicly available. With all these data repositories we share the concept of having diverse, automatically retrieved, software history data easily available on-line. However, with *SOFAS* is possible to proactively fetch such data for new projects, while these repositories handle only a fixed, pre-defined set of projects. Moreover, to the best of our knowledge, none of them have been used in any replication study.

The lack of proactiveness of data repositories can be overcome by tools and platforms combining a wide range of software analyses. Systems such as Kenyon [BWKG05], Evolizer [GFP09] and softChange [Ger04] can automatically extract and combine source code change history, bug history and additional analyses, such as, fine grained source code change extraction, source code meta-model reconstruction, etc. Moreover, they have means to make use of and interpret such data (e.g. visualizations, querying interfaces, etc.). These tools allow to easily replicate studies that are based exactly on the analyses provided. However, no other replication is possible as the supported analysis combinations are created beforehand and hardcoded into the tools. Furthermore, none of them address the issue of making the results easily available from outside of the tool, a crucial requirement for successful replication. The results are only available and can only be affectively read from the tool itself.

Gasser et al. [GRS04] point out the need for a sharable research infrastructure and collections of data under common access points and frameworks. In their paper, they also outline a general blueprint for such infrastructure. *SOFAS*, along with FOSSology [Gob08] and Alitheia Core [GS09] are systems devised to address that exact need. FOSSology is a framework to analyze source code with different, custom analyses (called agents) that can be created by users to fulfill their specific needs. The framework itself is just in charge of extracting the source code from a given repository which will then be analyzed by these analysis agents. However, as of now, the framework only presents an agent that detects the code license. Alitheia Core presents numerous similarities with *SOFAS*. They both are extensible platforms for software evolution analysis, integrating data collection facilities (from software repositories) with a varied assortment of analyses making use

of that data. Their main intention is to foster the creation of an extensible ecosystem of shared analyses and results for researchers to enrich and exploit. The main differences lie in the implementation (plug-in vs. service based architecture), the analyses already offered and how they can be combined. In fact, in contrast with *SOFAS*, Alitheia does not allow to freely combine analyses. Data plugins retrieve and process data from version control systems (Svn and Git), issue trackers (Bugzilla) and mailing lists. Metrics plugins then can be written to make use of such data to extract additional knowledge. However, these plugins cannot be further combined to provide more complex metrics nor their data be used by other plugins. Furthermore, the system does not provide any facility to allow users to effectively query such vast amount of data and extract additional information. Alitheia has already been used to extract the development data of more than 700 projects but the replication of existing studies has not been addressed.

The only analogous study existing in the literature, is the one performed by Tappolet at al. [JTB10]. In this study, the authors showed that if the data used by the MSR empirical studies were available in their software evolution ontology (EvoOnt), 75% of them could be reproduced with at most two SPARQL queries. However, no concrete study replication was performed. In fact, the goal of the authors was to demonstrate the potential of the inherent capabilities of the semantic web ontologies to better support software evolution research and overcome some of its most significant obstacles. We share with them the concept of representing software evolution data with ontologies and the opinion that they are extremely beneficial in the representation, sharing and combination of such data. However, we focus on the concrete replication of actual MSR studies and in proving how that can be addressed by a platform like *SOFAS* in concretely replicating such studies.

7.6 Conclusions

In this paper, we demonstrated how our *SOFAS* platform can be used to effectively replicate a fair amount of empirical studies previously presented at the Working Conference in Mining Software Repositories. To find these empirical studies, we performed a complete literature review of the papers published in the proceedings of any of the MSR conferences up to date (from 2004 to 2011). All the valid studies found were then classified into 6 different categories. For each of them, we manually assessed whether it could be replicated with a combination of the services currently found in *SOFAS*. As a result, we found that 30% of such studies could be fully replicated, while and additional 32% could be partially

replicated. A study was considered partially replicable if we could not replicate its results straight out of the box, but we could calculate all the ground data from which they are derived. At last, we presented in detail the concrete replication of two of these studies. The main purpose of it was to substantially demonstrate the feasibility of said replication and show how it is achieved with *SOFAS*.

The amount of studies that can be fully replicated is relatively low. However combined with the partially supported ones, we can simplify the replication of up to 62% of the currently existing empirical studies. In our opinion this is a promising results, as our main goal was to prove the applicability of a platform like ours in tackling the current issues hampering the replication in the software evolution community, namely uniform availability of analysis and results

The main limitation of our approach is that the breadth of replication support is limited by the analyses currently offered in *SOFAS*. Any further improvement would require the implementation and addition of new ones. In the future we plan to add additional analyses covering additional aspects of software evolution that have not been covered yet. Moreover, we plan to analyze a wide corpus of OSS projects with *SOFAS* to create on online data repository freely available to researchers to use as benchmark and to base their analyses on. We hope that the availability of such data combined with the results presented in this paper will spark interest in the research community to collaborate in *SOFAS* and, more in general, to tackle replicability in a more systematic and standardized way.

Acknowledgements

This work was supported by the Swiss National Science Foundation as part of the Systems of Systems Analysis (SoSYA) project, SNF Project No. 132175.

8

The Future of Software (Evolution) Analysis

An Architectural Blueprint for a Pluggable Version Control System for Software (Evolution) Analysis
Giacomo Ghezzi, Michael Würsch, Emanuel Giger and Harald Gall

Proc. Workshop on Developing Tools as Plug-ins (TOPI), 2012 (to appear)

CURRENT version control systems are not built to be systematically analyzed. They have greatly evolved since their first appearance, but their focus has always been towards supporting developers in forward engineering activities. Supporting the analysis of the development history has so far been neglected. A plethora of third party applications have been built to fill this gap. To extract the data needed, they use interfaces that were not built for that. Drawing from our experience in mining and analyzing version control repositories, we propose an architectural blueprint for a plug-in based version control system in which analyses can be directly plugged into it in a flexible and lightweight way, to support both developers and analysts. We show the potential of this approach in three usage scenarios and we also give some examples for these analysis plug-ins.

8.1 Introduction

The concept of managing the subsequent versions of the source code of a software project, and any other related document, has been around since the dawn of software development. The actual concept of a version control system (*VCS*) and its first implementation was introduced by Rochkind [Roc75] in the seventies. Systems belonging to this first generation were file-oriented, centralized, locking-based and without network access capability.

CVS [Ber90], an evolution of RCS [Tic85] paved the way for a second generation. It was explicitly designed for collaborative development and used a merging rather than locking-based approach. Through a client-server mode, geographically scattered developers were supported in working as a team.

The third and most recent generation represents yet another major conceptual shift: native complete decentralization. These systems quickly gained a remarkable popularity and ground over older, centralized systems, as dispersed, Internet-mediated software development became the norm rather than the exception. Well-known representatives of such distributed *VCSes* are Git and Mercurial.

As this very concise history shows, *VCSes* greatly evolved since their introduction. However, regardless of different implementation details and features provided, their core functionality and rationale never changed. No matter what *VCS* is being used, there are three basic things a user can do; *check out* a file copy from a repository, *check in* or *commit* a change on a file to its master in a repository, and *view the history* of files. Everything else is an elaboration or support for these three operations.

While the information stored in versioning systems supports traditional forward engineering activities sufficiently well, it is not complete enough to perform comprehensive evolution analysis or reverse. Recent research, however, has shown that there is much to be learnt from the development history of programs. The lack of support for such analyses has been filled so far by third-party tools that exploit *VCS* repository data to extract all sort of information, e.g. logical couplings, source code metrics, evolution of code clones, potential bugs, etc.

The only way such tools can retrieve this data has been the parsing and analysis of the bare history log; which is, in our opinion, far from being the optimal approach. These logs, in fact, record the history of a repository in a synthetic way and are meant mostly for users to keep track of the development history. Incremental, proactive processing is barely supported and retro-active computations are long, resource-intensive and often error prone. To put it in a nutshell, current *VCSes* are not built to be systematically analyzed. This

had a negative impact in the adoption of many software evolution analyses. In order for them to play a part in the developers' day-to-day processes and to prove their immediate usefulness, a more incremental, lightweight and integrated approach is needed, as pointed out by Zeller [Zel07].

In this paper we propose a plug-in-based *VCS*. Different analyses can be plugged into the system and register for specific repository events (e.g., a commit, the tagging of a new release, etc.). In this way, they can automatically and proactively run and update their data every time it is needed in an incremental fashion. Moreover, with the data produced, analyses can enrich the limited *VCS* data already existing. In the remainder of this paper, we first describe our proposed architectural blueprint. Next, we discuss the benefits of the architecture by comparing it to existing approaches in the context of three software evolution analysis scenarios. We then conclude with a brief discussion on future work.

8.2 Architecture Overview

Figure 8.1 gives a quick overview of the proposed architecture. At its core remains a standard *VCS*, offering all the functionalities of any modern, state of the art system. In fact, it is not our goal to propose a brand new *VCS*, as the current generation already supports forward engineering activities sufficiently well. Instead, we aim to enhance the existing ones by building a lightweight plug-in architecture on the top of them to remedy the lack of evolution analysis support in a flexible and transparent way. Most of the current *VCSes* already offer a mechanism to perform automated actions in response to specific events occurring in a repository. These actions are commonly called *hooks* or *triggers*. As of now, these mechanisms are underused and only for rather simple, low level tasks such as checking the compliance of commit messages, sending mail notifications, etc. In our architecture, these events are caught by the *Change Event Handler*, which uses them to build and maintain a detailed, high-level model of the history and state of the repository. This model, which we call *Version Control History Model*, describes all the essential concepts of a project's version control, independently of the actual *VCS* used. This is possible because most of the major version control systems share the same conceptual model, with just some slight differences in terminology. This is the model that the plug-ins see and use to fetch all the repository data they need. Once these events are processed and the model has been updated, an event containing the detailed change information is published to the *Evolution Event Notification Layer* which will notify the plug-ins. They

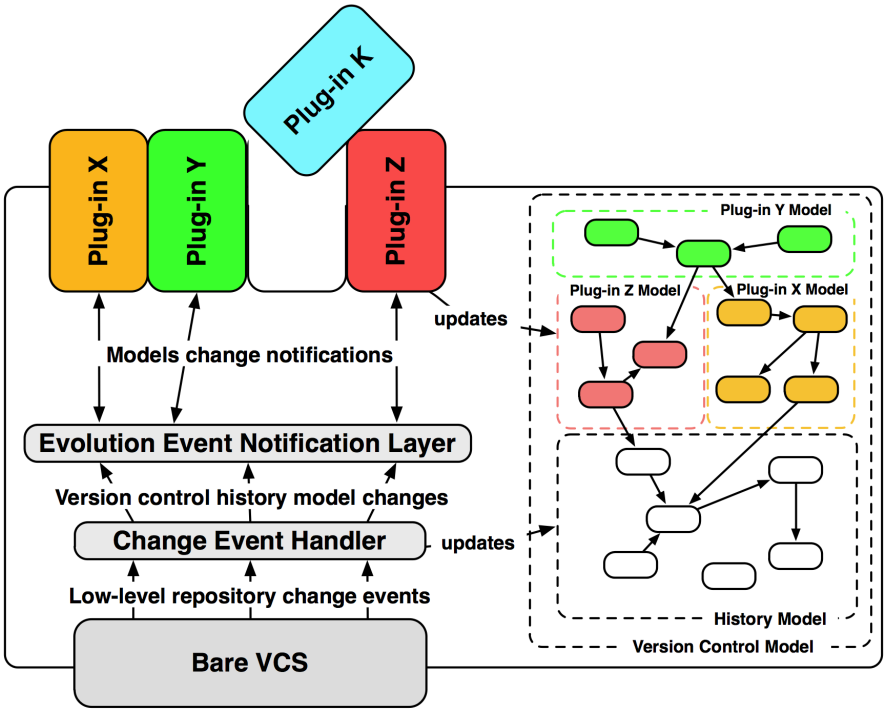

Figure 8.1: Overall view of the envisioned architecture.

will then use this information to run their analysis and/or update their data. Apart from consuming specific events, plug-ins can also directly query the model to extract further data. In fact, since an event only contains information on the entities directly involved in it, information about the past needs to be fetched directly from the model.

In the following we briefly describe each of the architectural components.

8.2.1 *Change Event Handler*

The *Change Event Handler* is the low level component through which our architecture connects to the actual *VCS*. *VCSes* produce several types of events, however, in our case we only catch the following:

- **Commit** occurs when a new change set is successfully committed in the repository.
- **Tag** occurs when a tag (also known as release) is created.
- **Branch** occurs after a new branch is created.

- **Merge** occurs when two branches or a branch and the main development trunk are merged together. It is basically a special commit.

The *Change Event Handler* extracts from these low level events all the information necessary to build and maintain the *Version Control History Model*. A high-level event reflecting the changes in the model is then created and published to the *Evolution Event Notification Layer*. These events contain the information about all the involved model entities. They can be considered a sort of translation of the repository events into a format that can be understood and used by the registered plug-ins.

8.2.2 *Version Control History Model*

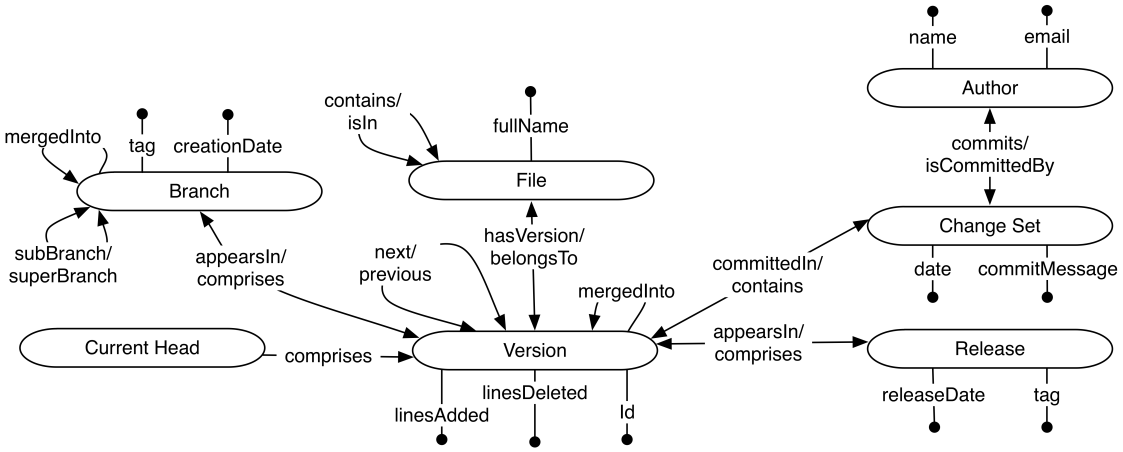

Figure 8.2: Overall view of the version control model.

The *Version Control History Model* is split into different parts. At the core of it lies the *History Model*, which describes all the core concepts of a version control history and is the part being managed by the *Change Event Handler*. This is the only part that is built into the system by default. Plugins can then expand and enrich it by defining on top of it their own sub-models to describe their analysis data. The main concepts of the *History Model* are:

- **Release** represents a snapshot in time of the project, labeled with a meaningful name or number. It thus comprises all the involved files versions consistently with the time of the snapshot (the most recent version, given the time of the snapshot). A new one

is created whenever a *tag* event is caught.

- **Branch** represents a branch of the project codebase at a specific point in time. In this way, both the branch and the original development stream can be worked on and evolve independently. A new one is created when a *branch* event is caught.
- **File** represents any file being tracked in the repository.
- **Version** is also known as Revision, it represents any type of change to a file under version control that was committed to the repository. It is uniquely associated with the file involved in the change. A new one is created whenever a *commit* event is caught, for every file involved.
- **Change Set** represents the set of changes on files that are written back to the repository at a specific moment in time by a user. It results in the creation of a new version of each modified file. A new one is created whenever a *commit* event is caught.
- **Author** represents a committer of the project.
- **Current Head** represents the current state of the main development trunk or stream. That is, the most recent versions of all its files.

Figure 8.2 gives an overview of this model. Hiding the repository events behind the *Change Event Handler* and using such a high-level model makes the entire architecture extremely flexible. In fact, it can be deployed on top of most *VCSes* by just providing a different *Change Event Handler* built to handle and interpret the system specific events. Moreover, the model structures repository data in a more intuitive and logical way. It can be directly queried by plug-ins, without them having to either analyze the repository history logs or its internal, low-level representation. Making the model extensible allows plug-ins to seamlessly enrich the original version control history data with their analysis data. Furthermore, it also enables them to benefit and use other plug-ins' data, thus supporting analyses not just based on the core historical data, but also on additional analysis data.

We define our model as an ontology with the Web Ontology Language OWL [De04] for three main reasons. First, a model described with such technology exhibits explicit semantics and is much more flexible to changes than one backed by a relational database. For example, it is unproblematic to extend ontologies by additions or by specializing existing concepts. On the other hand, a change in a traditional database-backed model

would usually require schema changes, which is a time consuming operation. Existing applications already accessing the database would likely break subsequent to the change. Second, ontologies were explicitly designed to be shared. They can be serialized using the RDF/XML standard and exchanged|in our case with the registered plug-ins|without any loss of data semantics. Third, a powerful and standardized language, SPARQL [PS08], can be used for querying.

8.2.3 *Evolution Event Notification Layer*

This component is in charge of sending the high-level events to the plug-ins. It is based on a publish-subscribe pattern, in which the subscribers are plug-ins and the publishers are the *Change Event Handler* and the plug-ins. Plugins can register for one or more types of events and react accordingly.

By default, only the four basic event types (*commit*, *tag*, *branch* and *merge*) are maintained by the system. Plugins, however, can register additional ones to publish any information about changes to their analysis data and thus notify any other possible plug-in that consume their data.

8.2.4 Plug-ins

Plug-ins are the consumers of the aforementioned events. They also query and extend the model while running analyses or offering additional functionality that extends and enriches that of the base *VCS*. Events only contain information on the entities directly involved in it. Further information, e.g., about the past of the entities then needs to be fetched from the model.

For example, a plug-in calculating version control history metrics, such as number of committed lines of code for each developer, activity by clock time, or the growth of the project's total lines of code over time, would just need the information about the new commit to update its data. On the other hand, a plug-in running a source code analysis, might need to get a snapshot of the entire project taken on the date of the new commit. Plug-ins do not necessarily just passively consume version control data for their own purpose. They can also expand the *Version Control History Model* with their own sub-model describing the data they produce. In this case, they can then also register new events representing the changes in their model and publish them to the *Evolution Event Notification Layer*.

8.2.5 An Operational Example

What follows is a quick outline of how our architecture and, in particular, its components react to a standard commit to the repository. A standard commit event is issued by a *VCS* every time changes on tracked files are successfully written back to the repository in an atomic operation. It always consists of a list of modified, added or deleted files, its unique version control ID or number and a message written by the author of the changes to describe them. Some systems also provide the number of added and deleted lines for every file involved (e.g. GIT), whereas others do not (e.g SVN). When the *Change Event Handler* catches a change event, a new *File* is created for every file that had just been added to the repository. A new *Version* is then created for each changed file and linked to its related *File*, as well as to the most recent previously committed *Version* (if there is any). If the committer does not yet exist in the model, a new *Author* is created, using all the information that can be extracted from the event. The *Versions* and the *Author* are then linked to a new *Change Set* representing the commit. The set will also contain the commit message, date and ID. Figure 8.3 shows the entities created and updated after a very simple commit. A new *commit* event with all these involved entities is then published to the *Evolution Event Notification Layer*.

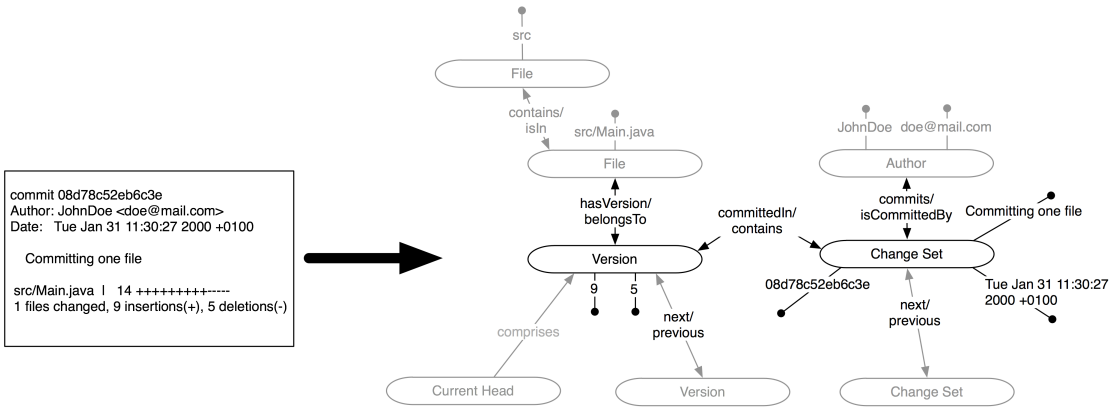

Figure 8.3: Handling of a new commit event.

8.3 Usage Scenarios

In the following, we list three problem scenarios in the context of software evolution analysis. For each scenario, we present existing solutions and their shortcomings; we then outline how our approach is able to overcome them with different plug-ins.

Scenario 1: Continuous Code Quality Check

Description: *Several studies proved that source code metrics are beneficial to steer the software development lifecycle. They are usually used to assess its overall quality [LM05, BBM96], discover problematic entities [FBB+99] or predict defects [GFS05]. Currently, VCSes only save and keep track of files. They do not discern between the different file types nor they analyze them. This means that, to calculate metrics, the entire source code needs to be fetched from the repository and parsed or even partially compiled.*

Existing Approaches: As of now, these metrics are calculated using third party tools. A snapshot of the project is manually checked out on a local machine and its source code fed to the metrics calculator of choice. Many IDEs have integrated calculators so that developers can check the metrics on their copies whenever needed. This approach however works only on local copies of the source code. Web-based software quality platforms, e.g., Sonar,[1] partially automate this process by fetching the source code to analyze directly from the repository upon user request. These systems can be triggered by Ant, Maven or Continuous Integration servers. This type of solution has proven to be highly successful and it is a big step in the direction of easy, automated software analysis. However, it requires installation and set up of a separate stack of heavy-weight applications, even for very simple measurements. Such an approach is also still not proactive, as the calculations have to be manually triggered by a user or a tool (e.g., a continuos integration application). Furthermore, if additional code quality indicators such as Code Smells or Disharmonies [LM05] need to be calculated, the exact same metrics will probably be recalculated again and again. In fact, all these tools are written to be used on their own and not to be combined or to share their data with each other. Even though these quality indicators are calculated using the exact same metrics already extracted, these synergies are lost.

Our Approach: A metrics plug-in is the only thing needed to address this scenario.

[1] http://www.sonarsource.org/

This plug-in subscribes to *commit* events published by the *Evolution Event Notification Layer* and proactively, continuously updates or calculates its metrics. Thus, at any point in time, the metrics data is always up to date. The speed of the analysis would also benefit. In fact the calculator works on very little changes (every commit) and accesses the files to analyze locally, without having to fetch them remotely and incurring in additional overhead. This plug-in could then also attach its own sub-model to the *Version Control History Model*, to describe the metrics it calculates and to relate them to the files under version control. In this way, an additional Code Disharmonies calculator could exploit this information to quickly keep track of all the suspect files exhibiting smells such a God Class, Brain Class, etc. Software engineers could either monitor these metrics by means of a Web front-end or with a plug-in for their IDE.

Scenario 2: Extracting Fine Grained Source Code Changes

Description: *VCSes still keep track of changes in a simplistic way, storing just the text lines that were added and/or deleted. Fine grained structural changes in the source code are not considered at all. Developers have to rely on textual diffs to really understand what and how code entities changed between different versions. Several studies already showed the usefulness of extracting and using such changes to detect re-factorings, discern different significance level of changes, better predict bugs, etc.*

Existing Approaches: In most of the cases, source code changes are extracted "a posteriori" given the entire *VCS* history for historical analysis [GFP09, KZJZ07]. This is extremely time consuming, as every single revision of every file has to be fetched from the repository and parsed to extract the information required. Moreover, these tools are not automatically triggered by changes in the *VCS* repository but have to be manually executed. Other tools have taken a more automated/proactive approach, extracting that information as changes are performed on a developer's local machine [RL08] or as they are committed to the repository [Ngu06]. So far, all these existing solutions are based on research prototypes and have not been incorporated in any of the commonly used *VCSes*.

Our Approach: A change extractor plug-in incrementally extracts these changes every time a new commit is performed. This plug-in responds to commit events through the *Evolution Event Notification Layer*. Every time one is received, it fetches the content of all the files involved and of their previous versions and calculates the fine-grained changes.

In this case, our system works much like Molhado [Ngu06], while being based on any already existing, well-known and widely used *VCSes*. Similar to the previous scenario, this incremental, proactive extraction of data is highly beneficial in terms of performance and ensures always up-to-date data.

Scenario 3: Flexible Querying of Custom Data

Description: *Modern VCSes do not have an interface through which information about them and their history can be programmatically extracted. The only way is to analyze their history logs, which have mainly been devised for record-keeping. They are intended to be read by human users and not suitable for systematic analyzes. Moreover, their format and syntax depends on the actual VCS. This means that every analysis, not only has to parse and interpret the log, but has to do that for every VCS addressed.*

Existing Approaches: All the major *VCSes* offer web interfaces both natively or through third-party tools. With these interfaces it is possible, for example, to see a list of all the files changed, added or deleted in any given revision or to compare two versions of a file manually to see what has been changed. These interfaces help human users to navigate the repository. An equivalent interface for applications, through which the repository can be queried for information, is still missing. Tappolet et al. [Tap08] introduced the concept of semantics-aware, queryable *VCSes*. However, to the best of out knowledge, it has never been implemented.

Our Approach: A *SPARQL Endpoint* plug-in allows users to query the internal version control history model with SPARQL [PS08] queries. This plug-in obviously needs to define and publish an ontology to describe that model, or use an existing one such as the ones introduced in [GG11]. This is necessary, so that users know the exact semantics and thus are able to write valid, meaningful queries. The plug-in then translates the SPARQL queries it receives into internal queries to fetch data from the internal model and then translate the results back into SPARQL Results [BB08]. In this way, from a user's perspective, the repository acts exactly like a RDF/OWL triple store. With our approach the query possibilities are manifold. Different query interfaces for different languages could be plugged into the system. For example, another plug-in could allow user to query the repository with natural language following the approach proposed by Würsch et al. [WGRG10].

8.4 Related Work

The idea of extending the functionality of a *VCS* is not new. Most of the currently existing systems already offer that. These extension mechanisms range from simple scripts only activated by specific repository events (e.g. SVN) to more complex plugin style solutions (e.g. Mercurial and Bazaar). These extensions can do a variety of things, including overriding commands, adding new commands, providing additional network transports, customizing log output, adding an alternative diff algorithm, etc. However, they are always aimed at extending or customizing the core functionalities of those systems. Our solution is not aimed at enriching those functionalities, but rather at building an infrastructure on top of a standard *VCS* to support its analysis. To the best of our knowledge such a solution has not been proposed yet.

There is a plethora of tools and frameworks exploiting software project data for all sorts of software evolution analysis. However, none of them are integrated within *VCSes*. Most of them, such as for example CodePro Analytix[2], require the installation of tools on a local machine and the manual triggering of such analyses. Sonar represents a step into a much more automated and continuous, plugin-based analysis engine for software projects. Nonetheless it is still not integrated with the targeted *VCSes* and it is mostly focused on software analysis (code coverage, test coverage, clone detection, etc.) and not on evolution.

We share with Molhado [Ngu06] the concept of an extensible, logical representational model to enrich the implicit version model used by standard *VCSes*. However, they exploit that to facilitate the tailoring of their proposed *VCS* to specific application domains. That is, they extend their base version control history model to support a more fine grained versioning of specific files. For example, on top of that, they built MolhadoRef, a *VCS* that supports the capturing and versioning of the semantics of Java program entities and refactoring operations that were performed on them. Our focus, on the other hand, is not on building a new, specialized *VCS* but on enhancing a standard one with pluggable analyses that can be transparently added and removed at any time.

[2] http://code.google.com/javadevtools/codepro/doc/index.html

8.5 Conclusions

In this paper we proposed an architectural blueprint of a new generation *VCS* that seamlessly supports software development and software (evolution) analysis. In our vision, evolution analyses should blend into *VCSes* in a transparent and lightweight way. We are confident that this could foster a broader use of evolution analyses during real software development and not just in the confines of academic research.

Based on our blueprint, we developed a first proof of concept prototype. This prototype features a stripped down version of all the presented architectural components and of two of the plug-ins we introduced earlier in Section 8.3: The SPARQL Endpoint and the Metrics Calculator. Building on this, we intend to proceed in developing a more sound prototype to be used in a case study. This would help us to further assess the strengths and weaknesses of our approach.

We use existing *VCSes*, as our purpose is to enhance the existing ones and not to reinvent the wheel. As for the *Version Control History Model*, we already have a fairly good knowledge in modeling and describing software evolution data with ontologies [GG11], which we will exploit and reuse for this project. The same goes for the analyses; in the next prototype we will create plug-ins out of the many analyses our group has developed throughout the years. These analyses range from OO metrics extractors, code disharmonies calculators, fine grained source code changes distillers, etc. A partial list can be found at `http://titan.ifi.uzh.ch/projects/sofas`.

9

Conclusions

Software evolution analysis is an effective way to support developers and other stakeholders in software maintenance and evolution tasks. In particular, it is key in having an always up to date and thorough view of a software system, its health and history. Such knowledge greatly helps in mitigate software aging and in reducing maintenance costs, which account for a large portion of the development effort and cost. Studies have highlighted the value of collecting and analyzing historical data stored into software repositories, such as version control, bug and issue tracking, or mailing lists for that purpose. A wide and growing range of different varied analysis techniques have been devised to collect and analyze this diverse sources of data. However, each of these techniques relies on its own methodologies and tools to extract, organize and utilize such data to produce the results needed. This means that, for every analysis, a specialized tool, with its own explicit or implicit meta-model dictating how to represent the input and the output, has to be installed, configured and executed. As a result, there is no way to compare or integrate the results of different analyses other than manual investigation, even when the analyses are conceptually of the same kind (*e.g.*, code duplication analysis). Interoperability is hampered even more by the stand-alone nature of such analyses, as well as their platform and language dependence.

Different languages and meta-models have been devised to make sense of such a wide ranged corpus of information and to consolidate it into known, well-structured, easily shareable representations. However, they have rarely been used by concrete analyses, outside use cases and proof of concepts. On the other hand, the issue of sharing and

integrating data has been ignored by the analyses and approaches proposed throughout the years. Data repositories and online analysis platforms represent some of the main efforts to fill this gap. Data repositories make all sort of software related historical data easily available online. Online platforms expose the analyses and their results online, using different interfaces. However, both approaches do not use any of the existing languages to describe such data, but they all use their own custom, often implicit, meta-models. Many times, these models are a basic transliteration of the data model of the databases in which the data is stored. Furthermore, the existing platforms do not support the systematical use of the exposed analyses through standard interfaces, e.g. web services, RMI, sockets, etc. Therefore, despite this richness of analyses, data representation languages and online platforms, software evolution analyses still suffer from three main problems. They are *rarely easy to re-use*, they exhibit *lack of clear and uniform data representation* and have *insufficient support for straight forward integration*.

In this thesis, we propose a solution to these problems by devising the concept of *Software Analysis as a Service* and *SOFAS*, the architecture implementing such concept. *SOFAS* is a distributed and collaborative software analysis platform that follows the principles of a RESTful architecture. It allows for interoperability of software evolution analyses across platform, geographical and organizational boundaries based on the principles of Representational State Transfer around resources on the web. Analyses expose their functionalities and data through standard RESTful web service interfaces and are mapped into a software analysis taxonomy. According to their category, they adhere to specific software analysis ontologies describing their input, output and the analysis itself. Their uniform interface enables their use over the Internet and their semi-automatic composition into complex analysis workflows. Such workflows can be used to ask specific questions regarding the evolution and the quality of software or to extract a wider range of data to fulfill broader and more open-ended information needs.

We claim that *SOFAS* can enhance and speed up the work of a software engineer by giving her access to a wide amount of information without the need to install several tools and to cope with many output formats. This data can be used as-is or as ground data by tools for further analysis and visualizations. It also promotes the uncovering of new, meaningful and interesting data deriving from the most diverse types of analysis that can finally "talk" to each other or can be combined in workflows. Moreover, such workflows can be then used to formulate and answer specific analysis question, such as *"What are the hotspots of a system?"*. At last, such a platform can also play a role in supporting the replication of empirical studies.

9.1 Summary of Results

The main contribution of this thesis is the concept of *Software Analysis as a Service* that we devised to solve the research problems we previously outlined, and *SOFAS*, the RESTful platform implementing it. The description of *SOFAS* is split between two separate works. In the first one we present its entire core architecture, its considerations and implementation aspects. In the second one we present in detail its software analysis composition component, together with the custom composition language we developed to define such composition. This work also presents a use case validation of *SOFAS* and its composition facilities. The solution we conceived to describe–using Semantic Web technologies–in a uniform and versatile way the data produced and consumed by the different software analyses is presented in a dedicated research work. In this work we also illustrate how such approach has been successfully use in two other different contexts, namely a natural language query interface for developers and large-scale software visualization. The natural language query interface is further described and evaluated in a dedicated work. At last, the entire approach is validated by proving its effectiveness in replicating existing empirical software evolution studies.

In the following, we summarize the goals and the results of each of these works.

1. **Software Analysis as a Service (Chapter 2, 33 et seq.).** The goal of this study is to lay down the theoretical foundation of this thesis. This include, framing the core research issues motivating our work and explain how we plan to solve them. As a result we:

   - Introduce the founding concept of *Software Analysis as a Service* aimed at solving such issues.
   - Propose a first architectural blueprint implementing such a concept.

2. ***SOFAS*, the Implementation of Software Analysis as a Service (Chapter 3, 53 et seq.).** The goal of this study is to devise the concrete architecture implementing the *Software Analysis as a Service* concept, as well as a set of ready-to-use services based on real usage scenarios. Moreover, we validate the architecture with a use case scenario and two concrete use cases. As a result we:

   - Develop and present in detail the proposed architecture for distributed analysis services based on the ideas sketched in previous work and built upon few initial experimental implementations. The proposed architecture follows the

principles of a REST [Fie00] and allows for a simple yet effective provisioning and use of software analyses based upon the principles of Representational State Transfer around resources on the web. It is made up by three main constituents: Software Analysis Web Services, a Software Analysis Broker, and Software Analysis Ontologies. Web services expose already existing analysis tools as standard RESTful web service interfaces. The Software Analysis Broker acts as the services manager and the interface between the services and the users. It contains a catalogue of all the registered analysis services with respect to a specific software analysis taxonomy. Software Analysis Ontologies define and represent the data consumed and produced by the different services.

- Introduce and briefly describe the Software Analysis Ontologies we use to describe the data produced and consumed by *SOFAS*' services.
- Evaluate the devised architecture with a use case scenario in which we show how a combination of *SOFAS*' services can support a user in a concrete software quality analysis task: finding the code smells of the major releases of ArgoUML[1].
- Further evaluate the approach with two concrete use cases of tools that are already making use of *SOFAS*. With such use cases, we aim to show *SOFAS*' versatility and usefulness.

3. **Ontologies, the Means to Describe the Product of Software Analysis Services (Chapter 4, 75 et seq.).** The goal of this study is to present in detail *SEON*, our family of Software Evolution Ontologies that we briefly introduced in the previous work. These ontologies describe software engineering and evolution knowledge on multiple levels of abstraction, ranging from code structures up to stakeholder activities. As a result, in this study we:

   - Critically reflect on the potential that the Semantic Web yields for *SOFAS* and software evolution in general. In particular, we show four characteristics that are most beneficial for the field: shared taxonomies, extensible meta-models, explicit relations, and Linked Data.
   - Validate *SEON* by describing three semantics-aware tools that make extensive use of it to help developers in dealing with large amounts of software evolution

[1] http://argouml.tigris.org/

data: *SOFAS*, a natural language query interface for developers and large-scale software visualization.

4. **Another Use of Ontologies in Software Engineering (Chapter 5, 109 et seq.).** The goal of this study is to present the framework to query for information about a software system using guided-input natural language resembling plain English that we introduced in the previous work. As a result we:

   - Describe in detail the framework proposed. Such system, integrated into Eclipse, allows software engineers to query for information about a software system using (quasi) natural language strongly resembling plain English. In the first proof of concept presented in this work, we focus only on supporting queries concerning static source code information, such as "How often is this field accessed?" or "What are the subclasses of this class?".
   - Provide a first case study evaluation of the approach, to demonstrate its potential. Evaluate the proposed approach with a case study in which we demonstrate, using the open source library JFreeChart[2] as an example, that it can be effectively used to answer the most common program comprehension questions that arise during software evolution tasks.

5. **A Composition Framework Built on Top of *SOFAS* and its Evaluation (Chapter 6, 135 et seq.).** The goal of this study is to present a novel framework for semi-automated software analysis composition built on top of *SOFAS*. This framework exploits the RESTful nature of *SOFAS* and comes with a service composer to enable semi-automated service compositions by a user. As a result we:

   - Explain how such composition works and introduce *SCoLa*, a new language we devised to define such a composition and model analysis workflows.
   - Present, as a proof of concept validation, two different approaches using such workflows to support different stakeholders in gaining a deeper insight into a project history and evolution. The first application conceptually proves that our framework can be used to address relevant evolution analysis questions, such as, finding code locations (i.e. hotspots) that have a high change frequency, intensive change coupling with other entities, and exhibit code clones. The

[2] http://www.jfree.org/jfreechart/

second application shows how tools can harness such workflows to automatically gather a wide range of varied yet interlinked information about a software system and how they can use that for their own specific needs. Both approaches were used during a quality assessment process of a commercial software we carried out with an industrial partner.

6. **Using *SOFAS* to Replicate MSR (Chapter 7, 173 et seq.).** We empirically evaluate the potential of *SOFAS* in replicating empirical studies on software evolution published throughout the years at the Working Conference on Mining Software Repositories (MSR[3]). As a result we:

   - Show that we can replicate, to different degrees of completeness, up to 62% of these studies. Studies that can be fully replicated account for 30% of the total, while the remaining 32% are studies that can only be partially replicated.
   - We replicate two of such MSR empirical studies and present in detail how the replication was carried out and the final results. The goal of this replication is to corroborate the replicability claims reported in the study and to better show the potential of *SOFAS* in such a replication context.

7. **A Possible Research Offshoot (Chapter 8, 201 et seq.).** In this study, we introduce one of the several future research directions inspired by this thesis. In particular, we propose an architectural blueprint for a plug-in based version control system in which software evolution analyses can be directly plugged into it in a flexible and lightweight way, to support both developers and analysts. We devised this architecture because currently existing version control system are not built to be systematically analyzed. Because of this, they are the major bottleneck and single point of failure in any analysis that uses version history. In order for such historical analyses to play a part in the developers' day-to-day processes, a new type of version control system is to be devised. As a result of this study we:

   - Describe in detail the proposed architectural blueprint in all its components
   - Discuss the benefits of our approach by comparing it to the current state of the art in the context of three software evolution analysis scenarios.
   - Introduce a first proof of concept prototype featuring a stripped down version of all the presented architectural components.

[3] www.msrconf.org

9.2 Implications of Results

Our proposed approach mainly affects to software engineering subfield: Mining Software Repositories and Software Evolution and Quality Analysis. In the following, we outline what we consider are the major implications of our work on those two fields.

Software Evolution and Quality Analysis The impact of *SOFAS* on this field is mainly demonstrated by the tools that are already using it. Some of these tools leverage different services registered in *SOFAS* to finally gain access to analyses that can produce the data they need in a clear and standard (both semantically and syntactically) format. Other tools, exploit *SOFAS* to its fullest by taking advantage of its composition functionalities to compose analyses into complex, meaningful workflows.

A Microsoft Surface application uses the data produced by a single service for purposes of multi-touch enabled code navigation and design recovery [MWS+12]. Another one, called SMELL TAGGER [MFGW12] uses the data produced by different services to detect and visualize the overall code structure, code smells [FBB+99] and multiple evolution metrics using different visualization paradigms (*e.g.*,, kiviat diagrams [PGFL05] and the house metaphor [BG07]) to support collaborative code review. Cocoviz [BG07] also uses the data produced by different software metrics services to visualize them using different cognitive metaphors and provide a better understanding of a software system and its state. All these tools have been successfully used to analyze several popular Java-based open source systems (*e.g.*,, ArgoUML, Eclipse, Vuze, jUnit, Tomcat, Derby). These tools demonstrate the usefulness of *SOFAS* in supporting users, in this case other applications, in the extraction and computation of valuable information from software repositories. Without a shared, easily accessible framework like *SOFAS*, these tools would have all had to implement the analyses needed on their own (this was indeed the case for COCOVIZ in its first versions). Furthermore, other external research groups have been using *SOFAS* services for different studies. For example, to investigate the factors of success and failure in open source projects and the contribution and collaboration patterns in OSS projects. These groups come from very different backgrounds, such as Physics, Management and Economics, proving how our approach can open the door to software evolution analysis to researchers other than software engineers. This, together with the fact that the different tools using *SOFAS* are developed with different programming languages and for different platforms, shows how our approach *allows for interoperability of software evolution analyses across platform, geographical and organizational boundaries.*

The ability to compose software evolution analyses into custom workflows is the most notable and prominent result of our approach. These workflows can either be used to fetch a mixture of specific, yet interlinked, evolution data for further analysis or visualization, or to answer precise analysis questions. Our Software Evolution Perspectives web application and *SOFAS* own composer web UI fully harness such aspect.

The composer UI guides human users in the composition of custom analysis workflows to answer specific questions. To prove this point, in this thesis we showed in detailed how it can be used to answer the question "Which are the hotspots and evolution anomalies for a project?". This question originates from a concrete need we encountered while performing a software quality audit of a commercial software with an industrial partner. The answer to such question consisted in a four lists of code entities that exhibited anomalies deemed problematic: high or abnormal values of known code quality metrics, known code smells, high change coupling and a lot of copied code (code clones). Moreover, a list of "super hotspots", entities presenting all the previous anomalies, was computed. Such data was used by the industrial partner to get an idea of the overall quality of its system and to pay particular attention and dedicate more resources to such problematic entities. Therefore, *SOFAS*, along with its composition language *SCoLa* and the composer web UI can be successfully used to address relevant, concrete, evolution analysis questions.

With such workflows, we can answer specific evolution analysis questions and single out, unequivocally, noteworthy bits of information. However, used on their own, they lack the capability to fulfill broader and more open-ended information needs. They provide all the information needed to fulfill those needs but, in this case, human interpretation is heavily needed to put everything into context and draw meaningful conclusions. Our Software Evolution Perspectives application enhances these workflows to fill such a gap. It automatically composes specific analysis workflows, using the specific *SOFAS* composer RESTful endpoints, to gather a wide range of varied yet interlinked information about a software system and present to the users a detailed and intuitive overview on its quality and history. It uses a combination of different "perspectives" focusing on different aspects of the software analyzed. Every perspective offers different interactive visualizations of the aspect addressed, along with automatically generated considerations about it, for users to better grasp the implications of the data being shown. Furthermore, based on such data, it also automatically compiles a written detailed report on the state of the analyzed project. Also this application has been used in the previously mentioned software quality audit. In this case, it helped our industrial partner to get a better sense of the overall quality of the system, its most problematic areas and to check whether specific quality indicators

improved or decreased with time. This was used to complement the more specific data about the system's hotspots and to put it into a broader picture. The automatically generated report, on the other hand, was used as a summary of the whole audit process and as a formal base for guiding the necessary improvements on the code base. This use case proves how the composition framework built on top of *SOFAS* can be effectively harnessed by tools to extract a wide range of interlinked evolutionary data and how such information is significant in an industrial context and not only for research.

Mining Software Repositories Having analyses as RESTful web services with a uniform interface greatly improves their accessibility. Users do not have to install or configure any tool, but just need to supply the analysis service with the necessary data. Moreover, services can also be easily integrated into custom user application and scripts. Being RESTful, they can be called with simple HTTP methods, without the need of custom libraries or frameworks. With this solution, also the results are available online, straight from the analysis that produced them. Furthermore, using public, well defined semantic web ontologies to describe these this broad, diverse data greatly facilitates its interpretation. Not only they describe in a clear way the domain of discourse, both semantically and syntactically, but they also come with a powerful, standard query language.

As demonstrated in our thesis, these features streamline and support the use and combination of analyses by both humans users and applications. Furthermore, they also make *SOFAS* a perfect candidate for supporting the replicability of empirical studies, as they fulfill the three main requirements for any successful replication. That is,

- **Availability of ground data.** The data on which the study is based should be easily and readily accessible in some form, preferably over the Internet.
- **Availability of the analysis itself.** The tools or scripts used to perform the study–which handle and analyze the ground data to produce the final results–should be publicly available and usable. If not, detailed instructions on how to perform the analysis, or even the algorithms, should be provided.
- **Availability and traceability of results.** The results produced in the study should be available in the same way as the ground data. This facilitates the verification of the results and claims of the original study and the comparison of the results of the replicas.

We corroborated such a claim by showing that we can replicate, to different degrees of

completeness, up to 62% of all the empirical studies published at the Working Conference on Mining Software Repositories.

Replication is a fundamental task in empirical studies and one of the main threats to validity that software evolution analysis (and, in general, empirical software engineering) may suffer. So far, this issue has been hardly addressed by the community. Moreover no systematic approach has been proposed. *SOFAS*, or a similar platform based on its principles, can provide a first viable solution to this issue. We do not claim this solution to be, in its current state, the definitive answer. However, it shows a first effective, uniform and lightweight approach that we hope would spark discussions on the topic and drive the community towards a more systematic approach to replication that is greatly needed.

9.3 Future Work

SOFAS provides several opportunities for future research. It offers a stable and working foundation for the software evolution analysis of the future, on which we and other research groups can further work. Moreover, many of its concepts can also be applied to other research contexts. In the following, we outline some of these new research opportunities:

- *SOFAS* is an active project and, by its own nature, in continuous evolution. New services offering brand new analyses or focusing on previously unsupported systems (*e.g.*, new version control system, issue trackers, etc.) are constantly added as soon as they are developed. This will continue also in the future. As of now, our research group is the main driving force behind it and the provider of the entire support infrastructure and services. In the future we plan to actively set up collaborations with other research groups to sharing their knowledge and their tools and thus provide new services to the architecture. This is vital in asserting the success and usefulness of our architecture, as one of its main foundations is indeed the sharing of new and diverse analyses by means of services.

- Recently, with the rise to prominence of cloud computing, a new generation of IDEs has emerged. These Ajax-based IDEs (such as, Cloud9[4], Coderun Studio[5] and Kodingen[6]) are based on a single shared instance running online and accessible from anywhere through any browser. This makes the coding environment always available,

[4] http://c9.io/

[5] www.coderun.com/

[6] www.kodingen.com

regardless of the location and platform being used by the developers. Furthermore, having a single shared instance, allows developers teams to easily collaborate in real time on the exact the same piece of code, as if they were in front of the same screen. Such IDEs only focus on supporting developers in standard coding tasks, but in-depth code and historical analyses are not yet supported. Due to their web-based nature, they could easily take advantage of *SOFAS* as the provider of a wide range of analysis data. For example, they could use such data to provide a family of code quality and evolution visualizations similar to our SOFTWARE EVOLUTION PERSPECTIVES (see Chapter 6, 135 et seq.). We plan, in the future to do a first pilot study in this direction using Cloud9.

- In Chapter 6 (135 et seq.), we offered some preliminary validations of *SOFAS* in its entirety. In particular, we demonstrated its value in a concrete, industry-based software quality audit. In the future, we plan do repeat similar case studies with other industrial partners in a more structured and rigorous way. In particular we plan to gather structured feedback from our partners to better assess and evaluate the strength and weaknesses of our approach.

- Our SOFTWARE EVOLUTION PERSPECTIVES application proved to be effective in offering different views over a software history and quality and in automatically generating audit reports based on that. We plan to expand it into a full-fledged software quality audit by: (1) implementing additional perspectives and visualizations, (2) expanding on its automatic report generation and (3) integrate well established software quality model, such as the SIG Maintainability Model [HKV07].

- We already showed how, through *SOFAS*' web UI, we can compose analyses into custom workflows to answer a specific evolution analysis question. In the future, we plan to gather a catalog of such questions/information needs through interviews with different stakeholders from an industrial partner that we have already been working with. We will then evaluate how well we can answer those questions using *SOFAS* workflows.

- In Chapter 8 (201 et seq.), we presented an architectural blueprint for a plug-in based version control system, along with a first proof of concept prototype. This approach was inspired by the experience in mining and analyzing version control repositories we gained while developing *SOFAS* and analyzing software projects with it. In particular, the fact that existing version control system are the major bottleneck and

single point of failure in any software evolution analyses that it is based on them. In the future, we plan to expand the current prototype, into a more sound and complete version, featuring several pre-registered analysis plug-ins to be used in a case study and to be publicly available for download.

- We plan to expand the replication study presented in Chapter 7 (173 et seq.) to empirical studies on software evolution published in all the major software engineering conferences and journals (*e.g.*, ICSE, ESEC/FSE, TSE and TOSEM).

A

Replication Study Queries

In the following we list the queries and the aggregations used in the replication presented in Section 7.4.

Find all the bug-introducing and bug fixing commits

This is accomplished with one query issued on the issue revision linker service page for the project being studied (urlhabanero.ifi.uzh.ch/bugFixesLinker/<projectname>) that fetches all the issues found together with the commits that introduced it and fixed it.

```
PREFIX f: <http://habanero.ifi.uzh.ch/seon/issuefixes.owl#>
PREFIX i: <http://habanero.ifi.uzh.ch/seon/issues.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?fix ?introduction
WHERE { ?i rdf:type i:Issue .
?fix f:fixes ?i .
?introduction f:caused ?i}
```

Extract the commit frequency and experience of the all the bug introducing users

This is achieved with a single query (issued on the issue revision linker service page for the project being studied, HABANERO.IFI.UZH.CH/BUGFIXESLINKER/<PROJECTNAME>) that fetches all the commits previously done by the authors of every buggy commit found at the previous step:

```
PREFIX v: <http://habanero.ifi.uzh.ch/seon/versions.owl#>
PREFIX t: <http://habanero.ifi.uzh.ch/seon/top.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?d ?buggyCommitDate
WHERE { ?a rdf:type t:Person .
?a v:commits ?c .
?c rdf:type v:ChangeSet .
?c v:hasCommitDate ?d .
# with the use of union (boolean OR) we can run
# a single query for all the buggy commits #
{BIND ("date of buggy commit 1" AS ?buggyCommitDate)} UNION
{BIND ("date of buggy commit 2" AS ?buggyCommitDate)} UNION
......
{BIND ("date of buggy commit n" AS ?buggyCommitDate)}
FILTER ( ?d < ?buggyCommitDate )}
# we the double ordering we are able to group the #
# commits by author and by date #
ORDER BY ASC (?buggyCommitDate) ASC(?d)
```

The results of this query can then be aggregated to get the information needed.

1. Get the first result of the query for every buggy commit and subtract it to the date of the buggy commit being considered to find the author experience at that time.

2. Sum the date difference between all the subsequent commits and divide it by the number of commits to get an average commit frequency of the author at that time.

Aggregate the buggy commits by time of the day, day of the week, developers experience and commit frequency

This is achieved with different queries and combination of their results, given the buggy commits found previously in section A. These queries have to issued on the specific project page of version control service, HABANERO.IFI.UZH.CH/BUGFIXES-LINKER/<PROJECTNAME>)

Aggregate buggy commits by the hour of the day

```
PREFIX v: <http://habanero.ifi.uzh.ch/seon/versions.owl#>
PREFIX t: <http://habanero.ifi.uzh.ch/seon/top.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?hour (COUNT(*) AS ?tot)
WHERE {
# we can extract the hour of every commit with #
# so standard string handling #
{BIND (substr("date of buggy commit 1", 12, 2) AS ?hour)}
    UNION
{BIND (substr("date of buggy commit 2", 12, 2) AS ?hour)}
    ......
{BIND (substr("date of buggy commit n", 12, 2) AS ?hour)}
} group by(?hour)
```

Aggregate buggy commits by the day of the week

```
PREFIX v: <http://habanero.ifi.uzh.ch/seon/versions.owl#>
PREFIX t: <http://habanero.ifi.uzh.ch/seon/top.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?dayName (COUNT(?dayID) AS ?tot)
WHERE {
```

```
# with the use of union (boolean OR) we can run a #
# single query for all the buggy commits #
    {BIND ("date of buggy commit 1"^^xsd:date AS ?date)}
    UNION
    {BIND ("date of buggy commit 2"^^xsd:date AS ?date)}
    UNION
    ......
    {BIND ("date of buggy commit n"^^xsd:date AS ?date)}
    BIND (day(?date) AS ?day)
    BIND (month(?date) AS ?month)
    BIND (year(?date) AS ?year)

    #Using the Gaussian algorithm to find day of the week#
    # Subtract 1 to year if January or February
    BIND (IF(?month<=2, 1, 0) AS ?jf)
    BIND (?year - ?jf AS ?adjyear)
    #century and year
    BIND (floor(?adjyear/100) as ?c) #?c is the century
    BIND (?adjyear - (?c * 100) as ?y) #?y is the year
    # x2 = (month + 9) % 12 + 1
    BIND (?month+9 AS ?x1)
    BIND (xsd:integer(?x1 - floor(?x1/12)*12 + 1) AS ?x2)
    # body of formula
    BIND ((?day + floor((2.6 * ?x2) - 0.2) + ?y +
              floor(?y/4) + floor(?c/4) - (2 * ?c)) AS ?i)
    BIND (?i - (floor(?i/7) * 7) AS ?dayIDx) # ?i % 7
    # ensure result is positive
    BIND (xsd:integer(IF
    (?dayIDx < 0, ?dayIDx + 7 , ?dayIDx))
              AS ?dayID)
   # Transform the results into a readable form #
    BIND (
      IF(?dayID = 0, "Sunday",
       IF(?dayID = 1, "Monday",
        IF(?dayID = 2, "Tuesday",
```

```
          IF(?dayID = 3, "Wednesday",
           IF(?dayID = 4, "Thursday",
            IF(?dayID = 5, "Friday",
             IF(?dayID = 6, "Saturday", "Unknown")
      ))))) AS ?dayName)
} GROUP BY (?dayName)
```

Aggregate buggy commits by developers experience and commit frequency This ground data was already extracted in the previous step (Section A), in which we calculated, for every single buggy commit, the experience and commit frequency of the developer who introduced it (calculated at the time of the introduction). In this step we just need to aggregate this data based on experience (in 4 months steps) and commit frequency (daily, weekly, monthly, other, single).

B

Studies Replication Summary

In the following, we briefly outline how we can replicate the MSR studies that we fully support with *SOFAS* (as reported in Section 7.3).

Paper title	**Services needed**	**Procedure**
Using CVS Historical Information to Understand How Students Develop Software. Y. Liu, E. Stroulia, K. Wong, D. German	• CVS importer	1. Extract the CVS history 2. Execute specific SPARQL queries directly on the data extracted to get the information needed

Continued on next page

Paper title	Services needed	Procedure
Mining Student CVS Repositories for Performance Indicators. K. Mierle, K. Laven, S. Roweis, G.Wilson	• CVS importer • Meta-model extractor • PMD service	1. Extract the CVS history and extract the needed features with SPARQL queries 2. Given the extracted CVS history, fetch the most up to date version of the code and extract its model, then find the needed features with SPARQL queries 3. Given the extracted CVS history, fetch the most up to date version of the code and feed it to the PMD service, then extract the needed features with SPARQL queries
Mining sequences of changed-files from version histories. H. Kagdi, S. Yusuf, J. I. Maletic	• Version control importer (CVS, git, SVN, etc.)	1. Extract the version control history 2. Run specific SPARQL queries to find relevant sequences of changed files

Continued on next page

Paper title	Services needed	Procedure
Mining Evolution Data of a Product Family. M. Fischer, J. Oberleitner, J. Ratzinger, H. C. Gall	• Version control importer (CVS, git, SVN, etc.) • Change coupling calculator	1. Extract the version control history 2. Run the needed lexical searches on the extracted history 3. Given the extracted history, calculate the change coupling between all the code entities and extract the needed information with SPARQL queries
An Exploratory Study of Identifier Renamings. L. M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y. G. Gueheneuc, G. Antoniol	• Version control importer (CVS, git, SVN, etc.) • Change Distiller	1. Extract the version control history 2. Given the extracted history, distill all the fine grained source code changes with the Change Distiller service 3. Extract the information needed with SPARQL queries

Continued on next page

Paper title	Services needed	Procedure
Fine Grained Indexing of Software Repositories to Support Impact Analysis. G. Canfora, L. Cerulo	• Version control importer (CVS, git, SVN, etc.) • Issue tracking importer • Issue-revision linker	1. Extract the version control history 2. Extract the issue tracking history 3. Reconstruct the links between issues and the revisions the fixed them 4. Extract the information needed with SPARQL queries and feed that to the impact analysis algorithm
Identifying Changed Source Code Lines from Version Repositories. G. Canfora, L. Cerulo, M. Di Penta	• Version control importer (CVS, git, SVN, etc.) • Change Distiller	1. Extract the version control history 2. Given the extracted history, distill all the fine grained source code changes with the Change Distiller service 3. Fetch all the information needed with SPARQL queries from the Change Distiller service

Continued on next page

Paper title	Services needed	Procedure
Mining Version Archives for Co-changed Lines. T. Zimmermann, S. Kim, A. Zeller, E. J. Whitehead	• Version control importer (CVS, git, SVN, etc.) • Change Distiller	1. Extract the version control history 2. Given the extracted history, distill all the fine grained source code changes with the Change Distiller service 3. Fetch all the information needed with SPARQL queries from the Change Distiller service
Understanding Source Code Evolution Using Abstract Syntax Tree Matching. I. Neamtiu, J. S. Foster, M. Hicks	• Version control importer (CVS, git, SVN, etc.) • Change Distiller	1. Extract the version control history 2. Given the extracted history, distill all the fine grained source code changes with the Change Distiller service 3. Fetch all the information needed with SPARQL queries from the Change Distiller service

Continued on next page

Paper title	Services needed	Procedure
Mining Email Social Networks. C. Bird, A. Gourley, P. Devanbu, M. Gertz, A. Swaminathan	• Version control importer (CVS, git, SVN, etc.) • GNU Mailman importer	1. Extract the version control history 2. Extract the entire email activity of the project 3. Fetch with SPARQL queries all the information needed to calculate the addressed social network measures 4. Retrieve with SPARQL queries developer information from the extracted history and correlate it with the social network data
Applying Social Network Analysis to the Information in CVS Repositories. L. Lopez-Fernandez, G. Robles, J. M. Gonzalez-Barahona	• CVS importer	1. Extract the version control history 2. Fetch with SPARQL queries all the information needed to calculate the addressed social network measures

Continued on next page

Paper title	Services needed	Procedure
Recommending Emergent Teams. S. Minto and G. C. Murphy	• CVS importer • Change coupling calculator • Code ownership detector	1. Extract the version control history 2. Given the extracted history, calculate the change coupling between all the code entities 3. Given the extracted history, calculate the code ownership of all the code entities 4. Fetch with SPARQL queries all the information needed from the change coupling and code ownership and combine it to calculate the authors' expertise
What can OSS mailing lists tell us? P. C. Rigby, A. E. Hassan	• GNU Mailman importer	1. Extract the entire email activity of the project 2. Fetch with SPARQL queries all the information necessary to calculate the measures needed

Continued on next page

Paper title	Services needed	Procedure
Using Software Repositories to Investigate Socio-technical Congruence in Development Projects. G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, C. Williams	• CVS importer	1. Extract the version control history 2. Fetch with SPARQL queries all the information needed to calculate the addressed social network measures
Security Versus Performance Bugs: A Case Study on Firefox. S. Zaman, B. Adams, A. E. Hassan	• Version control importer (CVS, git, SVN, etc.) • Issue tracking importer • Issue-revision linker	1. Extract the version control history 2. Extract the issue tracking history 3. Reconstruct the links between issues and the revisions the fixed them 4. Extract the information needed with SPARQL queries to calculate the necessary bug fixing metrics

Continued on next page

Paper title	Services needed	Procedure
Determining Implementation Expertise from Bug Reports. J. Anvik, G. C. Murphy	• Version control importer (CVS, git, SVN, etc.) • Issue tracking importer • Issue-revision linker	1. Extract the version control history 2. Extract the issue tracking history 3. Reconstruct the links between issues and the revisions the fixed them 4. Extract the information needed with SPARQL queries to calculate the two different implementation expertise models

Continued on next page

Paper title	Services needed	Procedure
Evaluating the Harmfulness of Cloning. A. Lozano, M. Wermelinger, B. Nuseibeh	• Version control importer (CVS, git, SVN, etc.) • Code clones history calculator • Change coupling calculator	1. Extract the version control history 2. Given the extracted history, calculate the history of the existing clones 3. Given the extracted history, calculate the change coupling between all the code entities 4. Extract and aggregate the information needed with SPARQL queries to check whether clones impact code changes and coupling

Continued on next page

Paper title	Services needed	Procedure
Do time of day and developer experience affect commit bugginess? J. Eyolfson, L. Tan, P. Lam	• Version control importer (CVS, git, SVN, etc.) • Issue tracking importer • Issue-revision linker	1. Extract the version control history 2. Extract the issue tracking history 3. Reconstruct the links between issues and the revisions the fixed them 4. Extract and aggregate the information needed with SPARQL queries to find out the distribution of fix buggy commits in the week and day and the influence of developers' experience

Continued on next page

Paper title	Services needed	Procedure
Clones: What is that Smell? F. Rahman, C. Bird, P. Devanbu	• Version control importer (CVS, git, SVN, etc.) • Issue tracking importer • Issue-revision linker • Code clones history calculator	1. Extract the version control history 2. Extract the issue tracking history 3. Reconstruct the links between issues and the revisions the fixed them 4. Given the extracted history, calculate the history of the existing clones 5. Extract and aggregate the information needed with SPARQL queries to check the impact of clones on bugs

Continued on next page

Paper title	Services needed	Procedure
Analysis of the Linux Kernel Evolution Using Code Clone Coverage. S. Livieri, Y. Higo, M. Matsushita, K. Inoue	• Version control importer (CVS, git, SVN, etc.) • Meta-model extractor • Size and complexity metrics calculator • Code clones calculator	1. Extract the version control history 2. Given the extracted history, extract the meta model (FAMIX) of every release 3. Calculate the size and complexity metrics of every release given its model 4. Given the extracted history, extract the clones of every release 5. Extract with SPARQL queries the data needed to calculate the code clone coverage of every release

Continued on next page

Paper title	Services needed	Procedure
Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones. M. Kim and D. Notkin	• Version control importer (CVS, git, SVN, etc.) • Code clones history calculator	1. Extract the version control history 2. Given the extracted history, extract the history of its clones 3. Extract and aggregate with SPARQL queries the data needed to reconstruct the genealogies and their status
When Do Changes Induce Fixes? J. Sliwerski, T. Zimmermann, A. Zeller	• Version control importer (CVS, git, SVN, etc.) • Issue tracking importer • Issue-revision linker	1. Extract the version control history 2. Extract the issue tracking history 3. Reconstruct the links between issues and the revisions the fixed them 4. Extract and aggregate the information needed with SPARQL queries to find out the distribution of fix inducing changes in the week

Continued on next page

Paper title	Services needed	Procedure
Predicting the Severity of a Reported Bug. A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals	• Issue tracking importer	1. Extract the issue tracking history 2. Extract with SPARQL queries allthe information needed to compare future bugs and predict their severity

Table B.1: The results of the replicability evaluation.

Bibliography

[ACPF01] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59:181–196, 2001.

[AHM06] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, 2006.

[AM07] J. Anvik and G. C. Murphy. Determining Implementation Expertise from Bug Reports. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 2, 2007.

[APM04] G. Antoniol, M. Di Penta, and E. Merlo. An automatic Approach to identify Class Evolution Discontinuities. *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 31–40, 2004.

[aVAdLM08] F. Dur ao, T. A. Vanderlei, E. S. Almeida, and S. R. de L. Meira. Applying a semantic layer in a source code search tool. In *Proceedings of the Symposium on Applied Computing*, 2008.

[AWR02] B. Kullbach A. Winter and V. Riediger. An Overview of the GXL Graph Exchange Language. In *Revised Lectures on Software Visualization*, pages 324–336. Springer-Verlag, 2002.

[AZ11] A. Alnusair and T. Zhao. Retrieving Reusable Software Components Using Enhanced Representation of Domain Knowledge. In *Recent Trends in Information Reuse and Integration*. Springer, Wien, Austria, 2011.

[Bak95] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, page 86, 1995.

[BB08] D. Beckett and J. Broekstra. SPARQL Query Results XML Format. W3C Recommendation, 15 January 2008. `http://www.w3.org/TR/rdf-sparql-XMLres/`.

[BBG+07] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *Software, IET*, 1(6):219–232, 2007.

[BBM96] V.R. Basili, L. C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):75– –761, 1996.

[BCF+07] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language, January 2007.

[Ber90] B. Berliner. CVS II: Parallelizing Software Development. In *Proc. USENIX Winter 1990 Technical Conference*, 1990.

[BG07] S. Boccuzzo and H. C. Gall. CocoViz: Towards Cognitive Software Visualizations. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 72–79, 2007.

[BGD+06] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143, 2006.

[BGD+07] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open Borders? Immigration in Open Source Projects. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 6, 2007.

[BJS+08] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, 2008.

[BKKK06] A. Bernstein, E. Kaufmann, C. Kaiser, and C. Kiefer. Ginseng: A Guided Input Natural Language Search Engine for Querying Ontologies. In *Jena User Conference*, May 2006.

[BKPS97] T. Ball, J. Kim, A. Porter, and H. Siy. If your version control system could talk. In *Proceedings of the International Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[BKZ10] A. Begel, Y. Phang Khoo, and T. Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd international conference on Software engineering*, pages 125–134, 2010.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396 - Uniform Resource Identifiers (URI). IETF RFC, August 1998. `http://www.ietf.org/rfc/rfc2396.txt`.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific America*, 284(5):34–43, 2001.

[BLR10] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *32nd International Conference on Software Engineering*, volume 1, pages 375–384, 2010.

[BNM+11] C. Bird, N. Nagappan, B. Murphy, H. C. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.

[Boe76] B. Boehm. Software Engineering. *IEEE Transactions on Software Engineering*, pages 1226–1241, Dec 1976.

[BPD+08] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 24–35, 2008.

[bpm11] Business Process Model And Notation (BPMN). OMG Standard, 2011. `http://www.omg.org/spec/BPMN/2.0/`.

[BRW+08] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller. Replication's Role in Software Engineering. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 365–379. Springer London, 2008.

[BSL99] V.R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, jul/aug 1999.

[BVG06] M.F. Bertoa, A. Vallecillo, and F. Garcia. An Ontology for Software Measurement. In *Ontologies for Software Engineering and Software Technology*. Springer, 2006.

[BW03] J. Bevan and E. J. Whitehead. Identification of Software Instabilities. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 134, 2003.

[BWKG05] J. Bevan, E. J. Whitehead, S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 177–186, 2005.

[BYM+98] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, pages 368–377, 1998.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. w3c note. W3C Recommendation, March 2001.

[CGK98] Y. F. Chen, E. R. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. *IEEE Transactions on Software Engineering*, 24(9):682–694, 1998.

[Cho04] G. G. Chowdhury. *Introduction to Modern Information Retrieval*. Facet, London, 2nd edition, 2004.

[CK94] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CMR92] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, 1992.

[Cre97] R. F. Crew. Astlog: A language for examining abstract syntax trees. In *In Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 229–242, 1997.

[dAM08] B. de Alwis and G. C. Murphy. Answering conceptual queries with Ferret. In *Proceedings of the 30th international conference on Software engineering*, pages 21–30, 2008.

[DBS91] P. Devanbu, R. Brachman, and P. G. Selfridge. Lassie: a knowledge-based software information system. *Communications of the ACM*, 34(5):34–49, 1991.

[DDN00] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 166–177, 2000.

[De04] M. Dean and G. Schreiber eds. *OWL Web Ontology Language Reference*. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/owl-ref/`.

[DE05] J. Dietrich and C. Elgar. A formal description of design patterns using OWL. In *Proceedings of the Australian Software Engineering Conference*, pages 243–250, 2005.

[DGLP08] M. D'Ambros, H. C. Gall, M. Lanza, and M. Pinzger. Analyzing Software Repositories to understand Software Evolution. In *Software Evolution*. Springer, Heidelberg, Germany, 2008.

[DLG05] M. D'Ambros, M. Lanza, and H. Gall. Fractal Figures: Visualizing Development Effort for CVS Entities. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, 2005.

[DLL09] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing Co-Change Information with the Evolution Radar. *IEEE Transactions on Software Engineering*, 99:720–735, 2009.

[DLR09] M. D'Ambros, M. Lanza, and R. Robbes. On the Relationship Between Change Coupling and Software Defects. In *16th Working Conference on Reverse Engineering*, pages 135–144, 2009.

[DLR10] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *7th IEEE Working Conference on Mining Software Repositories*, pages 31–41, 2010.

[DP03] D. Draheim and L. Pekacki. Process-Centric Analytical Processing of Version Control Data. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 131, 2003.

[dRW04] J. des Rivières and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[EAP+11] L. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y. Gueheneuc, and G. Antoniol. An exploratory study of identifier renamings. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 33–42, New York, NY, USA, 2011. ACM.

[EF95] J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. In *Graphtheoretic Concepts in Computer Science*, pages 38–50. Springer, 1995.

[EGK+11] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. In *IEEE Transaction on Software Engineering*, volume 27, pages 1–12, 2011.

[Erl00] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional (IEEE)*, pages 17–23, may/jun 2000.

[ETL11] J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 153–162, 2011.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[FG06] B. Fluri and H. C. Gall. Classifying Change Types for Qualifying Change Couplings. In *Proceedings of the 14th International Conference on Program Comprehension*, pages 35–45, 2006.

[Fie00] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[FKO98] L. Feijs, R. Krikhaar, and R. Van Ommering. A relational approach to support software architecture analysis. *Software Practice and Experience*, 28(4):371–400, April 1998.

[FL07] J. Farrell and H. Lausen. Semantic Annotations for WSDL and XML Schema. W3C Recommendation, 28 August 2007. `http://www.w3.org/TR/sawsdl/`.

[FPG03a] M. Fischer, M. Pinzger, and H. C. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th International Conference on Software Maintenance*, pages 23–32, 2003.

[FPG03b] M. Fischer, M. Pinzger, and H.C. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 19th International Conference on Software Maintenance*, pages 23–32, 2003.

[FUMK03] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of Web service compositions. In *Proceedings of th 18th IEEE International Conference on Automated Software Engineering*, pages 152–161, 2003.

[FWPG07] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change Distilling–Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

[GAM96] W.G. Griswold, D. Atkinson, and C. McCurdy. Fast, Flexible Syntactic Pattern Matching and Processing. In *4th International Workshop on Program Comprehension*, pages 144–153, 1996.

[GBO07] T. Menzies G. Boetticher and T. Ostrand. PROMISE Repository of empirical software engineering data. West Virginia University, Department of Computer Science, http://promisedata.org/repository 2007.

[GDL04] T. Girba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 40–49, 2004.

[Ger04] D. German. Mining CVS repositories, the softChange experience. In *Proceedings ot the International Workshop on Mining Software Repositories*, pages 17–21, 2004.

[GFP09] H. C. Gall, B. Fluri, and M. Pinzger. Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33, January/February 2009.

[GFS05] T. Gyimothy, R. Ferenc, and I. Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering*, 2005.

[GG08] G. Ghezzi and H. C. Gall. Towards Software Analysis as a Service. In *Proceedings of Evol'08, the 4th Intl. ERCIM Workshop on Software Evolution and Evolvability at the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering*, 2008.

[GG11] G. Ghezzi and H. C. Gall. SOFAS: A Lightweight Architecture for Software Analysis as a Service. In *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture*, 2011.

[GHJ98] H. C. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 190–198, 1998.

[GHM+07] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H.F. Nielsen, A. Karmarkar, and Y. Lafon. SOAP Version 1.2. W3C Recommendation, April 2007.

[Gin12] C. Gini. Variabilità e mutabilità. Memorie di metodologica statistica, 1912.

[GJK03] H.C. Gall, M. Jazayeri, and J. Krajewski. CVS Release History Data for Detecting Logical Couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 13=–23, 2003.

[GKSD05] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How Developers Drive Software Evolution. In *Proceedings of the 8th International Workshop on Principles of Software Evolution*, pages 113–122, 2005.

[GL02] M. Gruninger and J. Lee. Ontology Applications and Design. *Communications of the ACM*, 45(2):39–41, 2002.

[GMW00] D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, 2000.

[Gob08] R. Gobeille. The FOSSology project. In *5th IEEE International Working Conference on Mining Software Repositories*, pages 47–50, 2008.

[GPG10] E. Giger, M. Pinzger, and H. C. Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56, 2010.

[GPG11a] E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 83–92, 2011.

[GPG11b] E. Giger, M. Pinzger, and H. C. Gall. Using the gini coefficient for bug prediction in eclipse. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 51–55, 2011.

[GRS04] L. Gasser, G. Ripoche, and R. J. Sandusky. Research Infrastructure for Empirical Science of F/OSS. In *Proceedings of the International Workshop on Mining Software Repositories*, 2004.

[Gru93] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[GS09] G. Gousios and D. Spinellis. A platform for software engineering research. In *6th IEEE International Working Conference on Mining Software Repositories*, pages 31–40, 2009.

[GSZ04] Y.G. Guéhéneuc, H. A. Sahraoui, and F. Zaidi. Fingerprinting Design Patterns. *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 172–181, 2004.

[GZ05] Y.G. Guéhéneuc and T. Ziadi. Automated Reverse-Engineering of UML v2.0 Dynamic Models. In *Proceedings of the 6th ECOOP workshop on Object-Oriented Reengineering*, 2005.

[Had09] M. J. Hadley. Web Application Description Language (WADL). W3C Member Submission, 31 August 2009. `http://www.w3.org/Submission/wadl/`.

[Hal77] M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York, 1977.

[HBA08] G. Hughes, T. Bultan, and M. Alkhalaf. Client and server verification for web services using interface grammars. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 40–46, 2008.

[HCC06] J. Howison, M. Conklin, and K. Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, pages 17–26, 2006.

[Hen94] S. Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 11(5):48–59, 1994.

[HH05] A.E. Hassan and R.C. Holt. The Top Ten List: Dynamic Fault Prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005.

[HHHL03] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic Design Pattern Detection. *Proceedings of the 11th International Workshop on Program Comprehension*, pages 94–103, 2003.

[HKF08] O. Hartig, M. Kost, and J. C. Freytag. Automatic component selection with semantic technologies. In *International Workshop on Semamntic Web Enabled Software Engineering*, 2008.

[HKST06] H.J. Happel, A. Korthaus, S. Seedorf, and P. Tomczyk. KOntoR: An Ontology-enabled Approach to Software Reuse. In *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering*, 2006.

[HKV07] I. Heitlager, T. Kuipers, and J. Visser. A Practical Model for Measuring Maintainability. In *6th International Conference on the Quality of Information and Communications Technology*, pages 30–39, 2007.

[HKVD11] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 63–72, 2011.

[Hol98] R. C. Holt. Structural Manipulations of Software Architecture Using Tarski Relational Algebra. In *Proceedings of the Working Conference on Reverse Engineering*, page 210, 1998.

[HPSB+04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission, May 2004. `http://www.w3.org/Submission/SWRL/`.

[HPVS09] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, pages 232–242, 2009.

[HS04] S. J. Hespos and E. S. Spelke. Conceptual precursors to language. *Nature*, 430(6998):453–456, 2004.

[HS06] H. J. Happel and S. Seedorf. Applications of Ontologies in Software Engineering. In *2nd Workshop on Semantic Web Enabled Software Engineering*, 2006.

[HSP07] C. Hallett, D. Scott, and R. Power. Composing Questions through Conceptual Authoring. *Computer Linguistic*, 33:105–133, 2007.

[HT99] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley, Boston, MA, USA, 1999.

[htt] *j: A tool for dynamic analysis of java programs. http://www.sable.mcgill.ca/starj/.

[HVdM06] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 2–27, 2006.

[HWCK06] D. Hyland-Wood, D. Carrington, and S. Kaplan. Toward a Software Maintenance Methodology using Semantic Web Techniques. In *Proceedings of the 2nd International IEEE Workshop on Software Evolvability at IEEE International Conference on Software Maintenance*, pages 23–30, 2006.

[HWS00] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *Proceedings 7th Working Conference on Reverse Engineering*, page 162, 2000.

[Imb91] M. Imber. CASE data interchange format standards. *Information and Software Technology - Special issue on CASE (computer-aided software engineering)*, 33(9):647–655, November 1991.

[IUHT09] A. Iqbal, O. Ureche, M. Hausenblas, and G. Tummarello. LD2SD: Linked Data Driven Software Development. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pages 240–245, 2009.

[JC05] D. Jin and J. R. Cordy. Ontology-Based Software Analysis and Reengineering Tool Integration: the OASIS Service-Sharing Methodology. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 613–616, 2005.

[JE07a] D. Jordan and J. Evdemon. Web Services Business Process Execution Language Version 2.0. OASIS Standard, April 2007.

[JE07b] Diane Jordan and John Evdemon. Web services business process execution language version 2.0. OASIS Standard, 11 April 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[JKZ09] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120, 2009.

[Jon78] T. Capers Jones. Measuring Programming Quality and Productivity. *IBM Systems Journal*, 17(1):39–63, 1978.

[JR00] D. Jackson and M. Rinard. Software analysis: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering at ICSE 2000*, pages 133–145, 2000.

[JTB10] C. Kiefer J. Tappolet and A. Bernstein. Semantic web enabled software analysis. *Web Semantics*, 8(2-3):225–240, 2010.

[JV03] D. Janzen and K. De Volder. Navigating and Querying Code Without Getting Lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 178–187, 2003.

[KB07] E. Kaufmann and A. Bernstein. How Useful are Natural Language Interfaces to the Semantic Web for Casual End-users? In *6th International Semantic Web Conference*, pages 281–294, 2007.

[KBL08] C. Kiefer, A. Bernstein, and A. Locher. Adding data mining support to SPARQL via statistical relational learning methods. In *European Semantic Web Conference*, pages 478–492, 2008.

[KBS07] C. Kiefer, A. Bernstein, and M. Stocker. The Fundamentals of iSPARQL - A Virtual Triple Approach For Similarity-Based Semantic Web Tasks. In *Proceedings of the 6th International Semantic Web Conference*, pages 295–309, 2007.

[KBT07] C. Kiefer, A. Bernstein, and J. Tappolet. Mining Software Repositories with iSPAROL and a Software Evolution Ontology. In *Proceedings of the 2007 international workshop on Mining software repositories*, 2007.

[KC04] G. Klyne and J. J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

[KCM07] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution*, 19:77–131, 2007.

[KG01] R. Kollmann and M. Gogolla. Capturing Dynamic Program Behavior with UML Collaboration Diagrams. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, page 58, 2001.

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[KL95] E. Kamsties and C. M. Lott. An empirical evaluation of three defect-detection techniques. In *Proceedings of the Fifth European Software Engineering Conference*, pages 362–383, 1995.

[KN09] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 309–319, 2009.

[KP96] C. Krämer and L. Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-oriented Software. *Proceedings of the 3th Working Conference on Reverse Engineering*, pages 208–215, 1996.

[KPW05] S. Kim, K. Pan, and E.J. Whitehead. When functions change their names: automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 23–32, 2005.

[KRSR10] I. Keivanloo, L. Roostapour, P. Schugerl, and J. Rilling. Semantic Web-based Source Code Search. In *In Proceedings of the International Workshop on Semantic Web Enabled Software Engineering*, 2010.

[KSG+10] I. Kupershmidt, Q. J. Su, A. Grewal, S. Sundaresh, I. Halperin, J. Flynn, M. Shekar, H. Wang, J. Park, W. Cui, G. D. Wall, R. Wisotzkey, S. Alag, S. Akhtari, and M. Ronaghi. Ontology-Based Meta-Analysis of Global Collections of High-Throughput Public Data. *PLoS ONE*, 5(9), 2010.

[KSNM05] M. Kim, V. Sazawal, D. Notkin, and G.C. Murphy. An Empirical Study of Code Clones Genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 23–32, 2005.

[KWB05] S. Kim, E. J. Whitehead, and J. Bevan. Analysis of signature change patterns. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[KZJZ07] S. Kim, T. Zimmermann, E.J. Whitehead Jr., and A. Zeller. Predicting Faults from Cached History. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498, 2007.

[LB85] M. M. Lehman and L. Belady. *Program Evolution - Processes of Software Change*. London Academic Press, 1985.

[LBL06] J. Leduc, L.C. Briand, and Y. Labiche. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[LGS07] J. Lathem, K. Gomadam, and A.P. Sheth. SA-REST and (S)mashups : Adding Semantics to RESTful Services. In *International Conference on Semantic Computing*, pages 469–476, 2007.

[Lho07] O. Lhoták. Comparing call graphs. *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–42, 2007.

[LK94] M. Lorenz and J. Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[LM05] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[LSWG04] Y. Liu, E. Stroulia, K. Wong, and D. German. Using CVS Historical Information to Understand How Students Develop Software. In *Proceedings of the 2004 international workshop on Mining software repositories*, 2004.

[LTP04] T. C. Lethbridge, S. Tichelaar, and E. Plödereder. The Dagstuhl Middle Metamodel: a schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, pages 7–18, 2004.

[LZ05] B. Livshits and T. Zimmermann. Dynamine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 296–305, 2005.

[MAH10] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of ANT build systems. In *7th IEEE Working Conference on Mining Software Repositories*, pages 42–51, 2010.

[Man08] L. Mandel. Describe REST web services with WSDL 2.0, May 2008. `http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/`.

[Mar94] B. Marick. *Craft of Software Testing*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1994.

[McC76a] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2, 1976.

[McC76b] T.J. McCabe. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering*, page 407, 1976.

[McK84] J. McKee. Maintenance as a function of design. In *National computer Conference and Exposition*, pages 187–193, 1984.

[MFGW12] S. Müller, T. Fritz, H. C. Gall, and M. Würsch. An Approach for Collaborative Code Reviews Using Multi-touch Technology. In *5th International Workshop on Cooperative and Human Aspects of Software Engineering*, page to appear, 2012.

[MH02] A. Mockus and J.D. Herbsleb. Expertise Browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, 2002.

[Mil00] J. Miller. Applying Meta-Analytical Procedures to Software Engineering Experiments. *Journal of Systems and Software*, 54:29–39, 2000.

[MK88] H.A. Muller and K. Klashinsky. Rigi – A System for Programming-In-The-Large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, 1988.

[MKN09] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140, 2009.

[MLM96] J. Mayrand, C. Leblanc, and E.M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance*, pages 244–253, 1996.

[MM07] S. Minto and G. C. Murphy. Recommending Emergent Teams. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.

[Moc09] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *6th IEEE International Working Conference on Mining Software Repositories*, pages 11–20, 2009.

[mos08] MOST (Marrying Ontology and Software Technology) Project homepage, Last visited October 2008. `http://www.most-project.eu/`.

[MPS08] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

[MSW09] J. Mangler, E. Schikuta, and C. Witzany. Quo vadis interface definition languages? Towards a interface definition language for RESTful services. In

International Conference on Service-Oriented Computing and Applications, pages 1–4, 2009.

[MV00] A. Mockus and L. G. Votta. Identifying Reasons for Software Changes using Historic Databases. In *Proceedings of the 8th International Conference on Software Maintenance*, pages 120–130, 2000.

[MWG10] S. Demeyer M. Würsch, G. Reif and H.C. Gall. Fostering synergies - how semantic web technology could influence software repositories. In *2nd International Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, pages 45–48, 2010.

[MWM97] A. Brooks M. Wood, M. Roper and J. Miller. Comparing and combining software defect detection techniques: a replicated empirical study. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 262–277, 1997.

[MWS+12] S. Müller, M. Würsch, P. Schöni, G. Ghezzi, E. Giger, and H. C. Gall. Tangible Software Modeling with Multi-touch Technology. In *5th International Workshop on Cooperative and Human Aspects of Software Engineering*, page to appear, 2012.

[NB05] N. Nagappan and T. Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, 2005.

[Ngu06] T.N. Nguyen. Object-Oriented Software Configuration Management. In *22nd International Conference on Software Maintenance*, pages 351–354, 2006.

[NZ10] L. Nussbaum and S. Zacchiroli. The Ultimate Debian Database: Consolidating bazaar metadata for Quality Assurance and data mining. In *7th IEEE Working Conference on Mining Software Repositories*, pages 52–61, 2010.

[Obj98] Object Management Group. XML Metadata Interchange (XMI). Technical Report OMG Document ad/98-10-05, February 1998.

[OGS09] D. Oberle, S. Grimm, and S. Staab. An Ontology for Software. In *Handbook on Ontologies in Information Systems*. Springer, Heidelberg, Germany, 2nd edition, 2009.

[PA06] S. Pfleeger and J. Atlee. *Software Engineering - Theory and Practice*. Pearson Education, 3 edition, 2006.

[Par94] D. L. Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, 1994.

[Pau08] C. Pautasso. BPEL for REST. In *7th International Conference on Business Process Management*, pages 278–293, 2008.

[Pau09] C. Pautasso. Composing RESTful services with JOpera. In *International Conference on Software Composition*, pages 142–159, 2009.

[PGFL05] M. Pinzger, H. C. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of the ACM Symposium on Software Visualization*, pages 67–75, 2005.

[PGG07] M. Pinzger, E. Giger, and H. C. Gall. Handling Unresolved Method Bindings in Eclipse. Technical report, Department of Informatics, University of Zurich, Switzerland, 2007.

[PGKG08] M. Pinzger, K. Gräfenhain, P. Knab, and H. C. Gall. A Tool for Visual Understanding of Source Code Dependencies. In *Proceedings of the International Conference on Program Comprehension*, pages 254–259, 2008.

[PRSK10] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *IEEE International Conference on Software Maintenance*, pages 1–10, . 2010.

[PS08] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 15 January 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[PSE98] R. Power, D. Scott, and R. Evans. What you see is what you meant: Direct Knowledge Editing with Natural Language Feedback. In *Proc. Bienn. Eur. Conf. A.I*, pages 675–681, 1998.

[PSHe04] P. F. Patel-Schneider, P. Hayes, and I. Horrocks eds. *OWL Web Ontology Language Semantics and Abstract Syntax*. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/owl-semantics/`.

[RBD10] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *7th IEEE Working Conference on Mining Software Repositories*, pages 72–81, 2010.

[RC05] A. Rountev and B.H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *Proceedings of the 27th International Conference on Software Engineering*, pages 254–263, 2005.

[RD11] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500, 2011.

[RL08] R. Robbes and M. Lanza. Spyware: a change-aware development toolset. In *Proceedings of the 30th international conference on Software engineering*, pages 847–850, 2008.

[RN78] B. De Roze and T. Nyman. The software life cycle—a management and technological challenge in the department of defense. *IEEE Transactions on Software Engineering*, 4:309–318, July 1978.

[Rob10] G. Robles. Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories proceedings. In *7th IEEE Working Conference on Mining Software Repositories*, pages 171–180, 2010.

[Roc75] M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[SA96] P. Santanu and P. Ataul. A Query Algebra for Program Databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, 1996.

[SAG+06] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *IEEE Transactions on Software Engineering*, 7(32):454–466, 2006.

[SCM04] M.K. Smith, C.Welty, and D.L. McGuinness. OWL Web Ontology Language Guide. W3C Recommendation, February 2004.

[SCVJ08] F. Shull, J. C. Carver, S. Vegas, and N. Juristo Juzgado. The role of replications in Empirical Software Engineering. *Empirical Software Engineering*, 13(2):211–218, 2008.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

[SMV06] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34, 2006.

[SMV08] J. Sillito, G. C. Murphy, and K. De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[SNL+06] S.Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682, 2006.

[SPG+07] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[SWZ99] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES approach: language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, pages 487–550. World Scientific Publishing Co., Inc., 1999.

[Sys00] T. Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, Finland, 2000.

[SZZ05a] J. Sliwerski, T. Zimmermann, and A. Zeller. HATARI. Raising Risk Awareness. In *Proceedings of the 10th European Software Engineering Confer-*

ence held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 107–110, 2005.

[SZZ05b] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[Tap08] J. Tappolet. Semantics-aware Software Project Repositories. In *In Proceedings of the European Semantic Web Conference*, 2008.

[TDD00] S. Tichelaar, S. Ducasse, and S. Demeyer. FAMIX and XMI. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 296–298, 2000.

[THCS06] N. Tsantalis, S.T. Halkidis, A. Chatzigeorgiou, and G. Stephanides. Design Pattern Detection Using Similarity Scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.

[Tic85] W. F. Tichy. RCS - A System for Version Control. *Software Practice and Experience*, 15(7):637–654, July 1985.

[TKB10] J. Tappolet, C. Kiefer, and A. Bernstein. Semantic Web enabled software analysis. *Journal of Web Semantics*, 8(2-3):225–240, 2010.

[TP03] P. Tonella and A. Potrich. Reverse Engineering of the Interaction Diagrams from C++ Code. In *Proceedings of the International Conference on Software Maintenance*, pages 159–168, 2003.

[UJ96] M. Uschold and R. Jasper. A Framework for Understanding and Classifying Ontology Applications. In *Proceedings of IJCAI Workshop on Ontologies and Problem Solving Methods*, August 1996.

[uM03] D. Čubranić and G. C. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, 2003.

[uM04] D. Čubranić and G. C. Murphy. Automatic Bug Triage Using Text Categorization. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, 2004.

[VRHS+99] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 125–135, 1999.

[WD06] P. Weissgerber and S. Diehl. Identifying Refactorings from Source-Code Changes. *Proceedings of the 21th International Conference on Automated Software Engineering*, pages 231–240, 2006.

[Wel97] C. A. Welty. Augmenting abstract syntax trees for program understanding. In *Proceedings of the 12th international conference on Automated software engineering*, 1997.

[WGH+12] M. Würsch, G. Ghezzi, M. Hert, G. Reif, and H.C. Gall. Seon: A pyramid of ontologies for software evolution and its applications. *Computing*, pages 1–31, 2012.

[WGRG10] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting Developers with Natural Language Queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010.

[WHC05] P. Wagstrom, J. Herbsleb, and K. Carley. A social network approach to free/open source software simulation. In *In Proceedings of the First International Conference on Open Source Systems*, pages 16–23, 2005.

[WKR01] A.s Winter, B. Kullbach, and V. Riediger. An Overview of the GXL Graph Exchange Language. In *Revised Lectures on Software Visualization, International Seminar Dagstuhl Castle*, pages 324–336, London, UK, 2001. Springer-Verlag.

[WPZZ07] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How Long will it Take to Fix This Bug? In *Proceedings of the 4th International Workshop on Mining Software Repositories*, 2007.

[WS08] Chadd C. Williams and Jaime W. Spacco. Branching and merging in the repository. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 19–22, 2008.

[wsc02] Web services business process execution language version 2.0, 8 August 2002. http://www.w3.org/TR/wsci/.

[wsc05] Web services choreography description language, 9 November 2005. http://www.w3.org/TR/ws-cdl-10/.

[WZR07] R. Witte, Y. Zhang, and J. Rilling. Empowering Software Maintainers with Semantic Web Technologies. In *4th European Semantic Web Conference*, pages 37–52, 2007.

[Xmi07] Mof 2.0/xmi mapping, version 2.1.1. OMG, January 2007.

[XQW+10] Chunxiang Xu, Wanling Qu, Hanpin Wang, Zizhen Wang, and Xiaojuan Ban. A Petri Net-Based Method for Data Validation of Web Services Composition. In *IEEE 34th Annual Computer Software and Applications Conference*, pages 468–476, 2010.

[XS05] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2005.

[YMNCC04] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[YZY+08] L. Yu, J. Zhou, Y. Yi, P. Li, and Q. Wang. Ontology Model-Based Static Analysis on Java Programs. In *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 92–99, 2008.

[ZD09] H. Zhao and P. Doshi. Towards Automated RESTful Web Service Composition. In *International Conference on Web Services*, pages 189–196, 2009.

[Zel07] A. Zeller. The Future of Programming Environments: Integration, Synergy, and Assistance. In *Future of Software Engineering*, pages 316–325, 2007.

[ZG03] L. Zou and M.W. Godfrey. Detecting Merging and Splitting Using Origin Analysis. In *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003.

[ZNG+09] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100, 2009.

[ZW04] T. Zimmermann and P. Weißgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *1st International Workshop on Mining Software Repositories*, pages 2–6, 2004.

[ZWDZ04] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining Version History to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.

[ZWDZ05] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

Curriculum Vitae

Personal Details

Name	Giacomo Ghezzi
Date of Birth	January 25, 1982
Place of Birth	Milano, Italy
Citizenship	Italian

Education

2007 – 2012	*Phd* Department of Informatics University of Zurich, Switzerland
2004 – 2007	*MSc Degree in Computer Science* Politecnico di Milano, Italy and University of Illinois at Chicago, USA
2001 – 2004	*BSc Degree in Computer Science* Politecnico di Milano, Italy
1996 – 2001	*High School Degree* Liceo Scientifico Statale R. Donatelli, Milano, Italy
1993 – 1996	Middle School Milano, Italy
1988 – 1993	Elementary School Milano, Italy